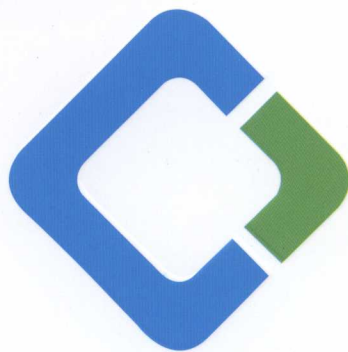


版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



Activiti: The Definitive Guide

Activiti

权威指南

冀正 张志祥 著

- 帮助读者快速熟悉Activiti源码，从而方便对Activiti源码进行扩展和修改，以满足业务需求。
- 指出Activiti框架中的部分Bug并修复。
- 对于复杂逻辑的讲解采用绘图以及剥洋葱方式阐述、层层分解，以此便于读者理解和掌握。
- 本书所有的实例均源于真实的企业应用，可移植性强、简单极致。

清华大学出版社



内容

Activiti: The Definitive Guide

权威指南

冀正 张志祥 著

- 讲解深入浅出、浅显易懂
- 实例来自实际的企业应用
- 对于 Activiti 框架、流程引擎、任务引擎、事件引擎、消息引擎、定时引擎、外部集成等都有详细的讲解
- 深入讲解 Activiti 的内部架构以及运行原理
- 一本“干货”书。

清华大学出版社
北京

内 容 简 介

本书从原理分析和企业应用两个方面,由浅入深,由易到难地对 Activiti 源码展开了系统深入的讲解,包括 Activiti 的底层架构设计思想以及缺陷修正、流程文档的解析、默认元素的解析、自定义元素的解析、事件转发器、事件清洗器、定时作业、流程虚拟机(PVM)、事务、缓存以及会话缓存、监听器、Activiti 封装 MyBatis 的整个过程、会签的实现(加签、退签和减签)、节点跳转(常规节点、分支节点、会签节点)、会签自定义权重实战、接管 Activiti 等内容。

本书不仅介绍了如何合理地使用 Activiti,还讲解了 Activiti 的使用误区和对框架中的部分缺陷进行修正以及优化扩展 Activiti 的技巧,从而使 Activiti 可以更好地为项目服务,帮助读者全面掌控和改造 Activiti。如果你想要深入透彻地掌控和改造 Activiti,那么这是你不可错过的一本好书。

本书的难度为中级到高级,适合于高校学生、所有的 Java 开发人员、工作流爱好者、Activiti 使用者、Flowable 学习者、研发人员、软件设计师、高级开发工程师和架构师等。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

Activiti 权威指南/冀正,张志祥著. —北京:清华大学出版社,2017

ISBN 978-7-302-47498-2

I. ①A… II. ①冀… ②张… III. ①JAVA 语言—程序设计—指南 IV. ①TP312.8-62

中国版本图书馆 CIP 数据核字(2017)第 142135 号

责任编辑:梁 颖

封面设计:李召霞 傅瑞学

责任校对:白 蕾

责任印制:杨 艳

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

课 件 下 载: <http://www.tup.com.cn>, 010-62795954

印 刷 者:北京富博印刷有限公司

装 订 者:北京市密云县京文制本装订厂

经 销:全国新华书店

开 本:185mm×260mm 印 张:25.25 字 数:612 千字

版 次:2017 年 8 月第 1 版 印 次:2017 年 8 月第 1 次印刷

印 数:1~2000

定 价:79.00 元

产品编号:071887-01

前言

Foreword

创作背景

对于 OA 内部办公系统、ERP 系统、审批系统等经常需要大量的使用工作流，而 Activiti 框架可以更好地辅助开发人员解决实际工作中遇到的流程问题，因此 Activiti 的重要性不言而喻。

很多朋友在实际项目开发中意识到了 Activiti 的重要性，也看了相关的书籍和资料，但是常听到这样的抱怨 Activiti 的书我看了不少，觉得看懂了，但就是不知道如何更好地扩展以及改造 Activiti，更不知道如何对其进行性能优化，达不到技术解决实际需求。

其实不难看出，造成这样的情况归根结底在于：这些朋友对于 Activiti 缺少架构方面的了解，缺少底层实现细节的认识，认为只使用 Activiti 提供的 API 就足够项目开发，其实还差得很远，并没有从更高层次来理解和认识 Activiti，没有“真正”让 Activiti 框架开足马力来为项目服务。正所谓了解其本质，掌握其原理才能更好地让 Activiti 框架为自身项目服务，而不是成为项目中的黑匣子、绊脚石。

市面上及网络上有不少 Activiti 方面的书籍，但对于一般的开发朋友而言，要么太深，不能拨云见日，要么太浅，看了跟没看没有什么区别，再次遇到类似的问题还是无从下手，其根本原因还是讲得太浅、不成系统，与企业开发脱轨。

对于所有想要深入学习 Activiti 框架的朋友，其实需要类似这样的书籍。

- 讲解深入浅出、浅显易懂。
- 实例来自实际的企业级项目开发，而不是来自虚拟的场景。
- 对于 Activiti 框架提供的功能，在开发过程中觉得部分功能不太好，可以灵活地扩展框架。
- 深入讲解 Activiti 的内部实现机制以及运行原理。
- 一本“干货”书。

这也是本书创建的目的，授人于鱼不如授之于渔，希望能够帮助更多的朋友更好地合理运用、扩展以及优化 Activiti，而不是仅仅局限于使用框架提供的 API。因为会用仅仅是入门而已，精通则是另一个层面的问题。

有过多年的开发经验以及一年的沉淀和准备最终成书，我们可以这样说：这是一本深入讲解 Activiti 的书，这是一本干货书，不仅有源码的讲解，还有基于源码讲解基础之上的实战讲解，这是一本诚意十足的书，这是一本深入讲解 Activiti 内核的书，这是一本真正的企业级实战书，敬请您研磨、品评鉴定。

本书的试读人员包括:在校大学生,1~10年的工作人员,职位覆盖中高级程序员、项目经理、系统架构师、总监、技术部经理和总经理等。

试读结果反馈:工作2年以下的开发人员,基本上都可以看明白,还需要认真研磨和消化吸收;工作2~5年的朋友,原来 Activiti 还可以这样用,真是满满的干货,达到了本书写作的目的和意图;工作5~8年的朋友,可以借鉴 Activiti 中的设计思想并运用到实际项目开发中;工作8~10年的朋友,这不仅是一本讲解如何使用 Activiti 的书,还讲解了 Activiti 框架的技术选型和架构思想的书,一本物有所值、货真价实的书。

本书体系结构

本书旨在深入学习 Activiti 的内部处理机制。由浅入深、由易到难地对 Activiti 源码展开系统地讲解和扩展,并对框架中的部分缺陷进行修正,从而可以使读者更加灵活地运用和改造 Activiti。

第1章 介绍 Activiti 环境搭建、设计器的安装、源码的获取和编译。

第2章 详细讲解流程配置文件的创建方式,流程引擎的架构、流程引擎配置类和流程引擎的创建内部实现机制,配置器的使用以及注意事项,流程引擎的管理以及流程引擎生命周期监听器的使用。

第3章 讲解流程资源的部署、各种部署方式以及 BpmnModel 与流程文档之间的转换和格式校验。

第4章 浓墨重彩地讲解流程文档解析原理和架构思想,包括文档转换器、流程元素解析、外围元素解析、根元素解析、流程元素解析、扩展元素解析、子元素解析、连线元素、黑名单机制以及元素解析器的架构设计。

第5章 详细讲解自定义元素的解析,包括自定义元素的解析原理、自定义元素的存储、黑名单元素以及非黑名单元素解析实战。

第6章 介绍事件转发器的内部实现机制,包括事件转发器的初始化、架构、注册方式,日志监听器的使用和扩展,日志清洗器的架构和自定义日志清洗器的使用。

第7章 深入剖析流程文档部署的内部实现机制,包括内置部署器的使用、部署命令以及自定义部署器实战。

第8章 讲解流程定义缓存的使用,包括自定义缓存处理类、节点缓存(新特性)的使用和 Bug 修复,以及配置器的高级使用技巧从而可以使流程信息彻底动态化。

第9章 讲解定时器的使用、运行原理以及自定义作业处理器实战。

第10章 讲解流程虚拟机的内部处理机制,包括对象解析器架构、自定义对象解析器、无缝入侵虚拟机以及操作表达式。

第11章 讲解监听器原理,包括监听器的生命周期,内置记录监听器,历史解析器的架构设计,监听器的触发时机和监听器代理类、自定义全局监听器的实现和字段注射模式(新特性)。

第12章 讲解 Activiti 中使用的命令模式,包括职责链模式,事务处理、事务上下文、事务监听器以及同步事务。

第13章 讲解流程虚拟机运转的整个过程以及各种原子类的功能边界和职责,包括异步与非异步节点处理和忽略节点功能。

第14章 讲解各种活动行为类的原理,包括排他网关行为类、自定义行为类,任务节点处理人多元化、多维度的处理,忽略节点使用误区以及 Bug 修复、子流程业务键 Bug 修复。

第 15 章 讲解 MyBatis 框架在 Activiti 中的使用,包括初始化数据源、数据访问层关系分析,自定义 Mapper 实战,Session 架构、SessionFactory 架构,SQL 语句适配器、SQL 执行 id 值生成规则,实体管理类,乐观锁,会话缓存的构建以及刷新过程。

第 16 章 实现高并发 id 生成器,讲解了变量类型的原理以及自定义变量处理类、ServiceLoader 方式注入配置器,任务的认领、归还、代理任务、会签的实现(加签、退签和减签),节点跳转(常规节点、分支节点和会签节点),会签自定义权重实战,接管 Activiti(映射文件、自定义部门实现、扩展任务节点参与者表和自定义代办 SQL),接管实体管理类。

本书约定

本书在讲述过程中,有如下约定:

- 引擎与流程引擎是等价的。
- 如无特殊说明,文档均指流程文档。
- 虚拟机默认指的是流程虚拟机
- 本书的程序、实例均在 JDK1.6 中运行,使用的数据库为 MySQL。

联系作者

本书的创作过程中可谓异常艰辛,由于 Activiti 内容涵盖面比较广泛,涉及的知识点非常之多,再加上 Activiti 框架中自身的一些缺陷和 Bug,因此为了能够让全书更加清楚、更加准确地阐述,笔者经历了许多不眠之夜。由于写作水平有限,本书不足之处在所难免,望读者谅解。更期待各界高手、专家就不足之处赐教。

为此,如果读者有任何的疑问或者建议,非常欢迎大家加入 QQ 群 129123599,一起探讨学习。我期待与大家一起交流学习、共同进步。同时也希望大家可以关注我的博客:<http://www.shareniu.com/>。

真诚致谢

创作的过程是痛苦的,持续的时间也远远超乎我的预期,本以为自己对 Activiti 已经了如指掌,但在写作的过程中还是会遇到各种各样的问题,幸运的是自己咬牙坚持下来了。

首先要感谢清华大学出版社的员工,本书的策划编辑,他们是我见过的最好的出版人,对本书从选题到出版的各个环节,都给予大量的指导和帮助,这对我的一生都有帮助。

其次要感谢我的妻子,一个产品经理,从产品的角度来让本书内容的层次感更加的清晰和人性化。她始终不辞辛劳,毫无怨言地对我照顾有加,才能我有更多的时间用来创作。

然后要感谢郑州轻工业学院张志锋教授、靳喜军、曾维林、李志、寇成星、张霞等在整个编写过程中给予的支持和帮助。

接下来,感谢家人、感谢朋友、感谢北京的明媚阳光和漫天风沙以及熙熙攘攘的回龙观,总之感谢一切。

最后,提前感谢购买本书的朋友们,您的支持信任是我们继续前进的动力。

冀 正

2017 年 5 月

目录

Contents

第 1 章 环境搭建	1
1.1 环境搭建	1
1.1.1 安装 GitHub	1
1.1.2 安装 Maven	1
1.2 编译源码	2
1.3 源码目录说明	3
1.4 安装流程设计器	4
1.5 工程搭建	5
第 2 章 探险流程引擎	7
2.1 流程配置文件	7
2.1.1 Activiti 配置风格	7
2.1.2 Spring 配置风格	8
2.2 流程引擎架构	10
2.3 构造流程引擎实例对象	12
2.3.1 初始化流程引擎之 Activiti 配置风格	14
2.3.2 构造流程引擎实例对象	15
2.3.3 创建流程引擎配置类实例	16
2.3.4 初始化流程引擎	18
2.3.5 初始化流程引擎之 Spring 配置风格	19
2.4 初始化流程引擎配置类	22
2.5 配置器	23
2.5.1 初始化配置器	23
2.5.2 配置器实战	25
2.6 初始化流程引擎	26
2.6.1 操作引擎表	28
2.7 管理流程引擎	29

2.7.1	注册流程引擎	29
2.7.2	关闭流程引擎	30
2.8	流程引擎生命周期监听器	30
2.9	其他方式构造引擎实例	31
2.9.1	ProcessEngineConfiguration 类创建引擎	31
2.9.2	编程方式创建引擎	33

第 3 章 初识流程资源部署 34

3.1	流程资源概述	34
3.1.1	流程文档部署生命周期	34
3.1.2	DeploymentBuilder 核心类	35
3.2	流程文档部署	36
3.2.1	定义流程文档	36
3.2.2	文本方式部署	37
3.2.3	classpath 资源部署	38
3.2.4	流式部署	39
3.3	BpmnModel 方式部署	40
3.4	校验 BpmnModel 实例对象	42
3.5	BpmnModel 转换流程文档	42
3.6	流程文档转换 BpmnModel	43
3.7	使用建议	43

第 4 章 流程文档解析原理 45

4.1	文档解析基础	45
4.1.1	文档解析模型	45
4.1.2	Activiti 文档解析技术选型演变	46
4.1.3	文档解析实战	46
4.2	元素解析功能架构设计	48
4.2.1	BPMN2.0 元素概述	48
4.2.2	元素解析功能架构设计	49
4.2.3	开闭原则	50
4.2.4	元素与元素属性承载类以及元素解析器的对应关系	51
4.2.5	元素属性承载类架构	52
4.3	元素解析环境准备	53
4.3.1	文档转换器	53
4.3.2	封装流程文档数据流	55
4.3.3	初始化元素解析器	56
4.3.4	文档转换器功能	57

4.3.5	元素解析环境准备	58
4.3.6	验证流程文档格式	60
4.4	元素解析	61
4.4.1	元素解析入口	61
4.4.2	解析根元素	64
4.4.3	流程内元素解析入口	66
4.4.4	解析连线	68
4.4.5	获取元素坐标	69
4.5	子元素解析	69
4.5.1	初始化子元素解析器	69
4.5.2	解析子元素	70
4.5.3	解析扩展元素	72
4.6	节点与连线关联	77
第 5 章	自定义元素解析	79
5.1	自定义元素解析原理	79
5.2	存储自定义元素属性值	80
5.3	自定义元素实战	81
5.4	扩展黑名单元素	83
5.4.1	扩展元素属性原理	84
5.4.2	任务节点扩展属性实战	84
5.5	扩展非黑名单元素	86
5.5.1	自定义元素解析器	86
5.5.2	替换引擎元素解析器	88
第 6 章	事件转发器	90
6.1	初始化事件转发器	90
6.2	事件转发器架构	91
6.3	注册事件监听器	93
6.4	事件转发功能之新老版本兼容	95
6.5	事件转发原理以及缺陷	96
6.6	添加事件监听器	98
6.6.1	使用配置方式添加	98
6.6.2	动态添加	99
6.7	日志监听器	100
6.7.1	初始化日志监听器	100
6.7.2	初始化日志处理器	101
6.7.3	日志处理器架构	102

6.7.4	收集日志数据入口	103
6.8	日志清洗器架构	106
6.8.1	数据库日志清洗器	106
6.8.2	生成日志数据	107
6.8.3	日志存储	108
6.9	自定义日志清洗器	108
第 7 章	流程文档部署原理	111
7.1	初始化部署器	111
7.1.1	初始化内置部署器	113
7.1.2	部署器依赖关系	115
7.2	部署命令	116
7.2.1	过滤重复文档	119
7.2.2	设置标识位	120
7.2.3	添加会话缓存	121
7.2.4	部署管理器	121
7.3	Bpmn 部署器	122
7.3.1	获取资源信息	126
7.3.2	封装资源信息	126
7.3.3	校验资源名称	127
7.3.4	计算流程定义版本值	128
7.3.5	生成流程定义 id 值	128
7.3.6	移除过期作业	129
7.3.7	添加作业	130
7.3.8	处理消息	130
7.3.9	处理信号	131
7.3.10	设置流程启动入	132
7.4	自定义部署器实战	133
第 8 章	缓存	134
8.1	背景	134
8.2	初始化缓存策略	135
8.3	部署管理器	136
8.4	缓存处理类架构	137
8.5	默认缓存处理类及 Bug	138
8.6	流程定义缓存	139
8.6.1	自定义缓存处理类	139
8.6.2	验证自定义缓存处理类	141

8.7	Activiti 新特性之节点缓存	142
8.8	节点缓存实战	142
8.9	节点缓存原理	144
8.9.1	初始化节点缓存数据	145
8.9.2	更新节点缓存	146
8.9.3	节点缓存架构	148
8.9.4	节点缓存使用误区	149
8.10	自定义节点缓存实战	150
8.10.1	自定义节点缓存类	150
8.10.2	修复 Activiti 节点缓存不更新 Bug	152
8.10.3	扩展引擎配置类功能	153
8.10.4	配置器高级用法	153
8.10.5	使用自定义节点缓存类	154
8.11	任务节点缓存数据获取原理	155
8.11.1	获取任务节点缓存数据	156
8.11.2	解析任务节点缓存数据	157
8.11.3	运用任务节点缓存数据	157
8.12	动态修改任务节点缓存数据	158
8.13	节点缓存使用技巧	159
第9章	定时作业	160
9.1	初始化作业执行器	160
9.2	初始化作业处理器	161
9.2.1	任务超时作业	163
9.2.2	定时任务作业	164
9.2.3	定时启动流程实例作业	165
9.2.4	其他作业	165
9.3	作业执行器原理	166
9.3.1	初始化作业执行器	166
9.3.2	启动作业执行器	167
9.4	添加定时作业	168
9.5	执行定时作业	170
9.6	处理作业	173
9.6.1	批量处理作业	174
9.6.2	执行作业之异常处理	177
9.7	关闭作业执行器	177
9.8	自定义作业处理器	178

第 10 章 流程虚拟机	181
10.1 流程虚拟机原理	181
10.2 虚拟机入口	183
10.3 流程定义转换准备	184
10.3.1 初始化对象解析器集合	185
10.3.2 初始化内置对象解析器	186
10.3.3 解析调度类 BpmnParseHandlers	188
10.3.4 BpmnParseHandler 架构	188
10.3.5 对象解析器架构	190
10.4 流程对象解析入口	191
10.5 流程子元素对象解析入口	194
10.5.1 任务节点对象解析器	195
10.5.2 创建 ActivityImpl 实例对象	196
10.5.3 多实例对象解析	199
10.5.4 连线对象解析	200
10.6 PvmProcessElement 类架构	201
10.7 自定义对象解析器	203
10.7.1 任务节点扩展属性	203
10.7.2 自定义任务节点对象解析器	204
10.7.3 获取自定义属性	206
10.7.4 运用自定义对象解析器	207
10.8 流程虚拟机实战	208
10.8.1 获取流程虚拟机对象	208
10.8.2 入侵流程虚拟机	209
10.9 操作连线表达式	210
10.9.1 自动计算连线表达式	210
10.9.2 获取连线表达式	211
第 11 章 监听器原理	212
11.1 监听器生命周期	213
11.2 内置记录监听器	214
11.2.1 内置任务记录监听器	214
11.2.2 内置执行记录监听器	215
11.3 历史解析器架构	215
11.3.1 添加内置记录监听器	216
11.3.2 初始化历史解析器	218
11.3.3 历史节点结束通知	219

11.3.4	控制归档历史数据级别	220
11.3.5	更新历史数据	221
11.3.6	历史节点开始通知	222
11.4	注入执行监听器	223
11.5	注入任务监听器	226
11.6	触发执行监听器	227
11.6.1	class 方式调度	227
11.6.2	delegateExpression 方式调度	230
11.6.3	expression 方式调度	231
11.6.4	执行监听器触发入口	232
11.7	触发任务监听器	233
11.8	监听器代理	234
11.8.1	默认代理类	234
11.8.2	自定义代理类	235
11.9	自定义全局任务监听器	236
11.10	Activiti 新特性之字段注射模式	238
第 12 章 Activiti 之设计模式		239
12.1	命令模式说明	239
12.1.1	命令模式的结构说明	239
12.1.2	命令模式实战	240
12.2	Activiti 命令模式	241
12.2.1	初始化命令配置类	242
12.2.2	Activiti 事务传播行为	243
12.2.3	Spring 事务拦截器	244
12.2.4	初始化命令调度者	245
12.2.5	初始化命令上下文工厂	245
12.2.6	初始化命令拦截器	246
12.3	Activiti 职责链模式	247
12.4	命令相关类职责	249
12.5	命令拦截器	250
12.5.1	日志拦截器	250
12.5.2	命令上下文拦截器	251
12.5.3	上下文类	252
12.5.4	创建命令上下文实例对象	253
12.5.5	命令调度者拦截器	254
12.6	自定义命令拦截器	255
12.7	命令类调度入口	256

12.8	Activiti 事务	257
12.8.1	MyBatis 事务管理	257
12.8.2	事务上下文架构	258
12.8.3	事务上下文工厂类	259
12.8.4	事务监听器	260
12.8.5	注册同步事务	261
第 13 章 流程虚拟机运转		262
13.1	流程实例运转入口	262
13.2	启动流程实例命令类	263
13.2.1	获取 ProcessDefinitionEntity 实例对象	265
13.2.2	重新生成流程定义缓存数据	265
13.3	创建流程实例	266
13.3.1	创建 ExecutionEntity 实例对象	269
13.3.2	获取 dataObject	272
13.3.3	区别流程实例与执行实例	272
13.3.4	添加历史流程实例数据	273
13.4	虚拟机运转原理	273
13.5	AtomicOperation 架构	274
13.6	流程实例启动	276
13.6.1	非异步节点处理	277
13.6.2	异步节点处理	278
13.7	原子类流转	281
13.7.1	流程启动原子类	281
13.7.2	流程启动准备原子类	282
13.7.3	活动节点执行原子类	283
13.7.4	开始节点行为类	283
13.7.5	途经连线	286
13.7.6	通知连线完成原子类	287
13.7.7	连线销毁原子类	287
13.7.8	其他原子类	289
13.8	Activiti 新特性之忽略节点	289
第 14 章 行为篇		291
14.1	活动行为工厂类	291
14.1.1	初始化活动行为工厂类	291
14.1.2	活动行为类架构	292
14.2	排他网关行为类原理	293

14.3	扩展排他网关实战	296
14.3.1	自定义排他网关行为类	296
14.3.2	自定义活动行为工厂类	297
14.3.3	替换默认活动行为工厂类	297
14.4	任务节点处理人多元化配置	298
14.4.1	任务处理人扩展	299
14.4.2	自定义任务解析器	299
14.4.3	自定义任务行为类	300
14.4.4	自定义活动行为工厂类	301
14.5	忽略节点使用误区	302
14.6	修复 Activiti 忽略节点 Bug	303
14.7	修复 Activiti 子流程业务键 Bug	303
第 15 章	Activiti 存储之 MyBatis	306
15.1	初始化 dataSource	306
15.2	Activiti 数据访问层关系分析	308
15.2.1	实体类与数据库表的映射	309
15.2.2	实例化 SqlSessionFactory	310
15.3	自定义 Mapper 实战	313
15.3.1	自定义 Mapper	313
15.3.2	自定义 SQL 执行原理	314
15.4	SessionFactory	316
15.4.1	初始化 SessionFactory	316
15.4.2	SessionFactory 架构	318
15.5	Session	319
15.5.1	Session 架构	319
15.5.2	实例化方式创建 Session 实例	320
15.5.3	反射方式创建 Session 实例	320
15.5.4	实例化 DbSqlSession	321
15.6	SQL 语句	321
15.6.1	SQL 语句适配器	321
15.6.2	SQL 执行 id 值生成规则	323
15.7	数据层和数据的关系	325
15.7.1	PersistentObject 业务对象	325
15.7.2	实体管理类	325
15.8	添加会话缓存	326
15.9	更新操作	329
15.9.1	会话缓存方式更新	329

15.9.2	SqlSession 方式更新	329
15.10	删除操作	330
15.10.1	DeleteOperation 接口	330
15.10.2	BulkDeleteOperation 删除数据	331
15.10.3	CheckedDeleteOperation 删除数据	332
15.10.4	乐观锁	334
15.11	刷新会话缓存入口	334
15.12	会话缓存数据持久化	336
15.12.1	移除不必要的数据库	336
15.12.2	刷新序列化变量	338
15.12.3	获取更新对象	340
15.12.4	刷新数据	342
15.12.5	解决依赖数据插入先后顺序	343
15.12.6	性能优化	344
第 16 章	实战	345
16.1	高并发 id 生成器	345
16.1.1	id 生成器初始化	345
16.1.2	自增 id 生成器	346
16.1.3	自定义主键生成器	347
16.2	变量类型	348
16.2.1	初始化变量管理类	348
16.2.2	变量管理类架构	350
16.2.3	变量处理类	350
16.2.4	自定义变量处理类	351
16.3	ServiceLoader 方式注入配置器	353
16.4	节点跳转	354
16.4.1	常规节点跳转	355
16.4.2	分支节点跳转	357
16.4.3	多实例节点跳转	359
16.5	会签	362
16.5.1	串行多实例	362
16.5.2	认领和归还任务	364
16.5.3	代理任务	364
16.5.4	并行多实例	365
16.5.5	加签	366
16.5.6	减签和退签	368
16.6	会签节点自定义权重实现	370

16.6.1	定义处理人权重	370
16.6.2	获取权重信息并自动计算	371
16.6.3	优化建议	373
16.7	接管 Activiti	374
16.7.1	接管 Activiti 映射文件	374
16.7.2	禁用用户表和组表	375
16.7.3	自定义用户角色和部门表	375
16.7.4	扩展任务节点参与者表	376
16.7.5	自定义任务节点参与者命令类	377
16.7.6	流程文档支持设置部门属性	378
16.7.7	解析及运用流程文档部门属性	379
16.7.8	自定义代办 SQL	380
16.8	接管 Activiti 实体管理类	381

1.1 环境搭建

由于 Activiti 将自己的代码交给 GitHub 托管,而且基于 Maven 构建项目,所以搭建 Activiti 源码环境首先需要安装 Maven 以及 GitHub。

1.1.1 安装 GitHub

可以从 GitHub 官网下载安装包,Windows 系统对应的版本下载地址为 <https://desktop.github.com/>。下载之后将其双击进行安装。安装成功之后,会出现 GitHub 的菜单,如图 1-1 所示。



图 1-1 GitHub 安装成功之后的菜单

1.1.2 安装 Maven

如果没有安装过 Maven,首先需要从 Maven 官方网站 <http://maven.apache.org/> 下载安装包,下载之后,将其解压到本地目录,然后需要修改环境变量 MAVEN_HOME 并将其添加到 PATH 环境变量中(形如 %MAVEN_HOME%\bin)。由于 Maven 依赖 Java 环境,因此使用 Maven 之前需要配置 Java 运行环境,否则 Maven 将不能正常使用。当以上操作执行完毕,在 Windows 控制台输入命令:mvn -v,如果安装成功则会显示 Maven 的版本信息,如图 1-2 所示。



图 1-2 Maven 安装成功之后的控制台输出

第 1 章

环境搭建

1.1 环境搭建

由于 Activiti 将自己的代码交给 GitHub 托管,而且基于 Maven 构建项目,所以要搭建 Activiti 源码环境首先需要安装 Maven 以及 GitHub。

1.1.1 安装 GitHub

可以从 GitHub 官网下载安装包,Windows 系统对应的版本下载地址为 <https://desktop.github.com/>,下载之后将其双击进行安装。安装成功之后,会出现 GitHub 的菜单,如图 1-1 所示。

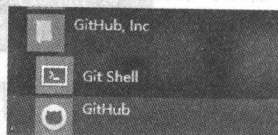


图 1-1 GitHub 安装成功之后的菜单

1.1.2 安装 Maven

如果没有安装过 Maven,首先需要从 Maven 官方网站 <http://maven.apache.org/> 下载安装包,下载之后,将其解压到本地目录,然后新建环境变量 MAVEN_HOME 并将其添加到 PATH 环境变量中(形如 %MAVEN_HOME%\bin)由于 Maven 依赖 Java 环境,因此使用 Maven 之前需要配置 Java 运行环境,否则 Maven 将无法正常使用。当以上操作执行完毕,在 Windows 控制台输入命令:mvn -v,如果安装成功则会显示 Maven 的版本信息,如图 1-2 所示。

```
C:\Users\ksf>mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-11T00:00:00Z)
Maven home: C:\software\apache-maven-3.3.9\bin\..
Java version: 1.7.0_80, vendor: Oracle Corporation
Java home: C:\software\Java\jdk1.7.0_80\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 8.1", version: "6.3", arch: "amd64", family: "windows"
```

图 1-2 Maven 安装成功之后的控制台输出

建议

JDK 最好使用 1.6 及以上版本。

1.2 编译源码

首先需要打开 GitHub,单击图 1-1 中的 Git Shell 选项,打开之后,需要设置期望将源码下载到本地目录,该操作需要使用 `cd` 命令手动进行,比如想要将下载的源码存储到 `E:\activiti` 下,则可以执行命令: `cd E:\activiti`,接下来输入以下命令:

```
git clone https://github.com/Activiti/Activiti.git
```

其中, `https://github.com/Activiti/Activiti.git` 为 Activiti 的源码地址,执行以上命令之后便开始进行源码下载,如图 1-3 所示。

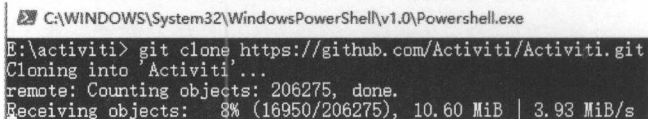


图 1-3 使用 GitHub 下载 Activiti 源码

源码下载可能比较耗时,经过一段时间的耐心等待,窗口状态如图 1-4 所示。

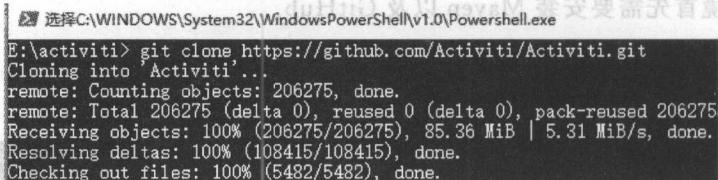


图 1-4 Activiti 源码下载之后的窗口显示

这时进入目录 `E:\activiti`,该目录已经存在了下载的源码,源码目录如图 1-5 所示,相关说明如下。

- (1) `distro`: 使用 Ant 工具下载文档资料以及将 Activiti 项目打包。
- (2) `README.md`: 记录 Activiti 团队的 JIRA 和 QA 地址。
- (3) `eclipse`: Activiti 团队使用的 Eclipse 模板文件。
- (4) `modules`: 该文件夹下存储了 Activiti 项目所有模块的 Java 源文件。



图 1-5 下载的 Activiti 源码

- (5) `qa`: 一些通用的流程配置文件样例。
- (6) `scripts`: Linux 平台下的一些启动脚本文件。
- (7) `userguide`: 用户操作手册,需要使用 Asciidoctor 工具生成。
- (8) `pom.xml`: 所有 Maven 工程的 parent。Activiti 工程依赖的第三程序包均定义在该文件中。

获取源码之后,接下来就是对源码进行编译。首先打开控制台,然后在控制台中输入命令 `cd E:\activiti`,最后输入命令 `mvn install -Dmaven.test.skip=true`,该过程可能非常耗费时间,经过一段时间之后,如果不出意外,控制台的输出如图 1-6 所示。

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Activiti ..... SUCCESS [ 37.375 s]
[INFO] Activiti - BPMN Model ..... SUCCESS [ 2.812 s]
[INFO] Activiti - Process Validation ..... SUCCESS [ 1.303 s]
[INFO] Activiti - BPMN Layout ..... SUCCESS [ 0.845 s]
[INFO] Activiti - Image Generator ..... SUCCESS [ 1.199 s]
[INFO] Activiti - BPMN Converter ..... SUCCESS [ 2.033 s]
[INFO] Activiti - Engine ..... SUCCESS [ 14.546 s]
[INFO] BUILD SUCCESS
```

图 1-6 Activiti 源码编译成功

这时再次进入图 1-5 modules 文件中的任意一个子目录(Activiti 工程模块)就会发现,已经有了 Eclipse 工程需要的 .classpath 和 .project 文件,如图 1-7 所示。接下来打开 Eclipse 工具,将工程导入,如图 1-8 所示。

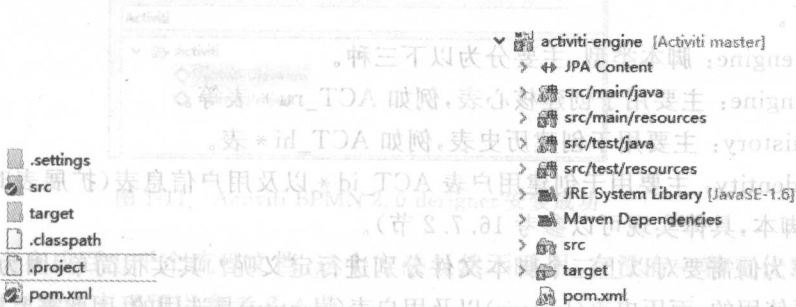


图 1-7 Eclipse 可以识别的 Activiti 源码

图 1-8 导入 Eclipse 后的源码工程

下面对 activiti-engine 的源码工程图进行相关说明。

- (1) `src/main/java`: 用于存放项目工程的核心实现逻辑代码。
- (2) `src/main/resources`: 用于存储配置文件。
- (3) `src/test/java`: 对核心代码进行单元测试。
- (4) `src/test/resources`: 存储对核心代码进行单元测试时需要使用的配置文件。

注意

Activiti 更多版本的下载可以进入 Activiti 官网 <http://www.activiti.org/download.html>, Activiti 项目的 GitHub 网站为 <https://github.com/Activiti/Activiti>。

1.3 源码目录说明

Activiti 源码下载之后,可以查看 modules\activiti-engine 模块中的 `src\main\resources` 文件夹,该文件夹中存储了流程引擎操作数据库需要的 DDL 脚本,如图 1-9 所示。

- (1) `create`: 创建数据库、表视图、索引的脚本。形如 `activiti.db2.create.engine.sql`,其

中 db2 代表操作的数据库类型。

(2) drop: 删除数据库、表视图、索引的脚本。形如
activiti.db2.drop.engine.sql。

(3) upgrade: 升级数据库、表视图、索引的脚本。形如
activiti.db2.upgradestep.53.to.54.engine.sql, 其中 53 对应
引擎的版本(5.13), 54 对应引擎需要升级到的版本(5.14)。

(4) mapping: 由于 Activiti 底层使用 MyBatis 框架操作数据库, 因此该文件下对应
MyBatis 框架需要的所有的映射文件。

以上目录中定义的脚本文件规范如下, 这里以 activiti.db2.create.history.sql 脚本为
例进行讲解。

(1) activiti: activiti 工作流引擎的标识。

(2) db2: 数据库厂商名称, 可以是 MySQL、H2、Oracle、PostgreSQL 等, 比如本书中使
用的是 MySQL 数据库, 那么操作时选择 activiti.mysql 开头的脚本文件即可。

(3) create: 创建数据库表的操作, 对应有 drop、upgradestep 标识符号, 操作时选择对
应的即可。

(4) engine: 脚本类型, 主要分为以下三种。

- engine: 主要用于创建核心表, 例如 ACT_ru* 表等。
- history: 主要用于创建历史表, 例如 ACT_hi* 表。
- identity: 主要用于创建用户表 ACT_id* 以及用户信息表(扩展表时可以不使用该
脚本, 具体实现可以参考 16.7.2 节)。

引擎为何需要对以上三个脚本文件分别进行定义呢? 其实很简单, 因为核心表(engine)
是必须要使用的, 而历史表(history)以及用户表(identity)是选用的, 因此需要加以区分。

1.4 安装流程设计器

本书使用的 Eclipse 版本为 Luna Release (4.4.0), Activiti 版本为 5.21.0。流程设计器为
Activiti BPMN 2.0 designer, 其官方下载地址为 <http://www.activiti.org/designer/archived/>,
可以进入该网站下载指定版本, 本书使用的 Activiti BPMN 2.0 designer 版本为 5.18。

对于 Activiti BPMN 2.0 designer 插件的安装建议采用在线安装的方式, 首先打开
Eclipse, 单击菜单 Help, 然后单击“Install New Software”, 再单击 Add 按钮, 填写信息如
图 1-10 所示, 最后单击 OK 按钮。

经过一段时间的耐心等待, 如果不出意外, 安装成功之后就可以建立 Activiti 工程了,
依次单击菜单 File→New→Other, 如图 1-11 所示。

Activiti BPMN 2.0 designer 在 5.8 版本之前流程设计文档的后缀为 .activiti, 比如流
程文档名称为 sharenui.activiti, 则保存该文件之后会自动生成 sharenui.bpmn20.xml, 5.9
版本之后, Activiti 废弃了这一做法, 流程文档的后缀修改为 .bpmn, 看到这里的处理, 可能
会有疑问: 如何兼容这两个版本的流程文档? 最简单的方法, 直接将 sharenui.bpmn20.xml 文
件名称修改为 sharenui.bpmn, 然后将 sharenui.activiti 文件删除即可。因为这两个文件的
内容都是相同的, 只是文件的后缀不同而已, 这种方法只适用于项目中流程文档比较少的情

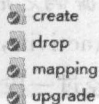


图 1-9 Activiti 中的 DDL 脚本

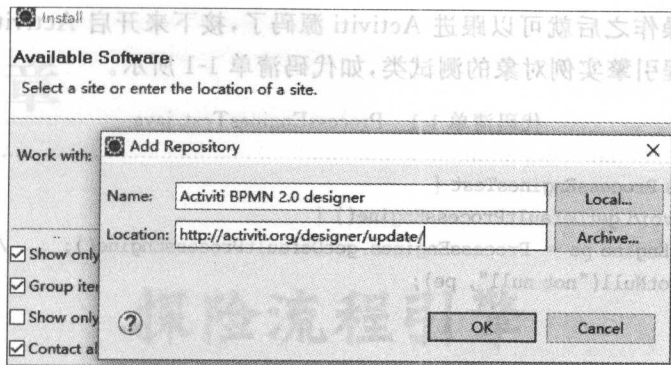


图 1-10 Eclipse 在线安装 Activiti BPMN 2.0 designer

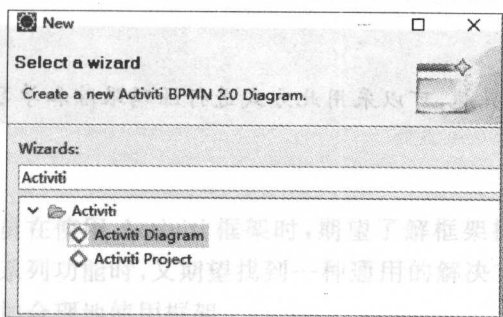


图 1-11 Activiti BPMN 2.0 designer 安装成功

情况下,如果项目中有成百上千个流程文档,这个方法显然不适用,有没有一种更优雅的方式解决这个问题呢?具体实现可以参考 7.4 节。

1.5 工程搭建

在全面学习 Activiti 源码之前,很有必要回顾一下 Activiti 最简单的用法:获取流程引擎实例对象。

首先建立一个名为 Activiti 的 Maven 工程,然后将 2.1 节中的 Activiti 风格配置文件 `activiti.cfg.xml` 放到目录 `src/main/resources` 中,在 Activiti 源码中,实现下面功能的是 `activiti-engine-5.21.0.jar`,因此进行源码查看时需要引入该工程,右击工程名(Activiti 项目),选择 `Properties`→`Java Build Path`→`Projects`,单击 `Add` 按钮,选择图 1-5 中的 `modules/activiti-engine` 项目即可,如图 1-12 所示。

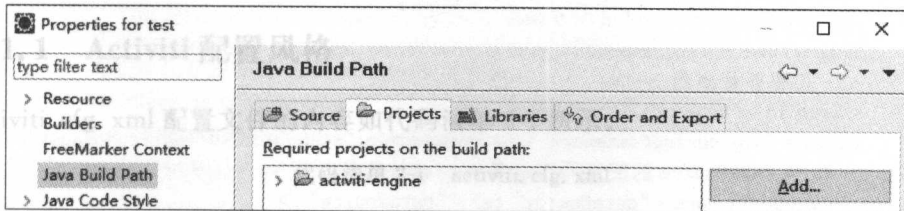


图 1-12 项目引入 Activiti 源码

代码清单 1-1 ProcessEnginesTest.java

代码清单 1-1 ProcessEnginesTest.java

```
1 public class ProcessEnginesTest {
2     public void getDefaultProcessEngine() {
3         ProcessEngine pe = ProcessEngines.getDefaultProcessEngine();    //获取流程引擎实例
4         assertNotNull("not null", pe);
5     }
6 }
```

运行上面的代码,如果程序不报错就可以看到控制台的相应输出结果。

扩展

如果对其他模块感兴趣,可以采用此方式进行源码跟踪和学习。

建築工程 5.1

第 2 章

探险流程引擎

一般情况下,开发人员在使用 Activiti 框架时,期望了解框架提供了哪些功能、如何使用、在使用框架提供的一系列功能时,又期望找到一种通用的解决方案,以便灵活地对框架提供的功能进行扩展,更加合理地使用框架。

流程引擎为开发人员提供了一系列扩展点,这些扩展点均可以通过流程引擎配置类进行操作。开发人员可以通过流程引擎配置类构造流程引擎实例对象,从而划清了流程引擎配置类与流程引擎类的使用边界,也使两个类的职责更加单一,因此本章以流程引擎的初始化为入口,详细讲解流程引擎的实现原理,从而解开流程引擎的神秘面纱。

2.1 流程配置文件

接下来重点分析流程引擎实例对象的创建过程。首先讲解在 Activiti 中如何定义流程配置文件,Activiti 中的流程配置文件类型可以分为以下两种。

- (1) 普通配置,即 Activiti 配置风格,通常情况下,使用该方式的文件名称为 activiti.cfg.xml。
- (2) Spring 配置,即 Spring 配置风格,通常情况下,使用该方式的文件名称为 activiti-context.xml。

以上两种方式均可实现流程引擎的配置工作,接下来具体分析这两种配置风格的实现。

2.1.1 Activiti 配置风格

activiti.cfg.xml 配置文件的内容如代码清单 2-1 所示。

代码清单 2-1 activiti.cfg.xml

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```



```

3      xsi:schemaLocation = "http://www.springframework.org/schema/beans
4      http://www.springframework.org/schema/beans/spring-beans.xsd">
5      <!-- 数据源配置 -->
6      <bean id = "dataSource" class = "com.alibaba.druid.pool.DruidDataSource">
7          <property name = "driverClassName">
8      public class <!-- 驱动类名称 -->
9      public <value>com.mysql.jdbc.Driver</value>
10     </property>
11     <property name = "url">
12     <!-- 数据库连接 URL -->
13     <value>jdbc:mysql://127.0.0.1:3306/shareniu</value>
14     </property>
15     <property name = "username"><!-- 数据库用户名 -->
16     <value>root</value>
17     </property>
18     <property name = "password" value = "shareniu" /><!-- 数据库密码 -->
19     </bean>
20     <!-- 流程引擎标准配置类 -->
21     <bean id = "processEngineConfiguration"
22         class = "org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration" >
23     <property name = "dataSource" ref = "dataSource" /><!-- 数据源 -->
24     <propertyname = "processEngineName" value = "shareniu" /><!-- 流程引擎名称 -->
25     </bean>
26 </beans>

```

通过上面配置文件的内容,可以看出 Activiti 配置风格本质上还是使用了 Spring 中的文件配置方式,上面的配置文件中看到了一系列 bean 的声明,尽管 Spring 中对于 bean 元素的定义有多种实现方式,但是上面的这种方式已经足够 Activiti 使用了。

代码清单 2-1 中的第 21~25 行定义了一个 id 值为 processEngineConfiguration 的流程引擎配置类,然后为其设置数据源和流程引擎名称两个属性值,到此为止 activiti.cfg.xml 文件的配置已经结束,上述配置已经足够本书讲解使用。

2.1.2 Spring 配置风格

activiti-context.xml 配置文件的内容如代码清单 2-2 所示。

代码清单 2-2 activiti-context.xml 内容

```

1 <beans xmlns = "http://www.springframework.org/schema/beans"
2     xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation = "http://www.springframework.org/schema/beans
4     http://www.springframework.org/schema/beans/spring-beans.xsd">
5     <!-- 事务管理器 -->
6     <bean id = "transactionManager"
7         class = "org.springframework.jdbc.datasource.DataSourceTransactionManager">
8         <!-- dataSource 的配置可以参考 Activiti 风格配置 -->
9         <property name = "dataSource" ref = "dataSource" />
10    </bean>
11    <!-- Spring 流程引擎配置类 -->

```

```

12 <bean id="processEngineConfiguration"
13     class="org.activiti.spring.SpringProcessEngineConfiguration">
14     <property name="dataSource" ref="dataSource" />
15     <property name="databaseSchemaUpdate" value="true" />
16     <property name="transactionManager" ref="transactionManager" />
17     <property name="deploymentResources" value="classpath *:./shareniu/au. *.
    bpmn20.xml"/>
18 </bean>
19 <!-- Spring 流程引擎 -->
20 <bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
21     <property name="processEngineConfiguration" ref="processEngineConfiguration" />
22 </bean>
23 <!-- 以下是各种流程引擎服务类 -->
24 <bean id="repositoryService" factory-bean="processEngine"
25     factory-method="getRepositoryService"/>
26 <bean id="runtimeService" factory-bean="processEngine"
27     factory-method="getRuntimeService"/>
28 <bean id="taskService" factory-bean="processEngine"
29     factory-method="getTaskService"/>
30 <bean id="historyService" factory-bean="processEngine"
31     factory-method="getHistoryService"/>
32 <bean id="managementService" factory-bean="processEngine"
33     factory-method="getManagementService"/>
34 </beans>

```

activiti-context.xml 的配置方式与 activiti.cfg.xml 的配置方式大体相似,因为两者本质上都是 Spring 配置方式。代码清单 2-2 中的第 6~10 行配置了事务管理器并为其设置数据源 dataSource,数据源的定义可以参照 activiti.cfg.xml 中的定义。第 12~18 行定义了流程引擎配置类,第 20~22 行定义了流程引擎类并为其设置了 processEngineConfiguration 属性值,如果使用 Spring 配置风格,则必须定义流程引擎类,否则构造流程引擎实例对象时程序会报错,该步骤非常重要,可以参考 2.3.5 节的讲解。接下来,定义一个 Spring 配置风格的测试类以验证流程引擎类是否被实例化,如代码清单 2-3 所示。

代码清单 2-3 ProcessEnginesTest.java

```

1 public class ProcessEngineTest {
2     public void testBean(){
3         //实例化 ApplicationContext 类
4         ApplicationContext ac = new GenericXmlApplicationContext("activiti-context.xml");
5         //获取 ProcessEngine 类型的实例对象
6         Map<String, ProcessEngine> beansOfType = ac.getBeansOfType(ProcessEngine.class);
7         //取出第一个 ProcessEngine 实例对象
8         ProcessEngine pe = beansOfType.values().iterator().next();
9         //获取任务服务类 TaskService 实例对象
10        TaskService taskService = pe.getTaskService();
11        assertNotNull("not null", taskService);
12        assertNotNull("not null", pe);
13    }
14 }

```

以上两种不同的配置方式使用了不同的流程引擎配置类,前者使用了标准的流程引擎配置类为 `StandaloneProcessEngineConfiguration`,后者使用的引擎配置类为 `SpringProcessEngineConfiguration`(该类位于 `activiti-spring-5.21.0.jar` 程序包中)。`SpringProcessEngineConfiguration` 类顾名思义,通常情况下与 Spring 框架整合时需要使用该类。

约定

本书如果没有特殊说明,则流程引擎配置类均为 `StandaloneProcessEngineConfiguration`,流程引擎类为 `ProcessEngine`。

2.2 流程引擎架构

Activiti 框架提供的流程引擎配置类 `ProcessEngineConfiguration` 的类图如图 2-1 所示,该类的架构如图 2-2 所示。

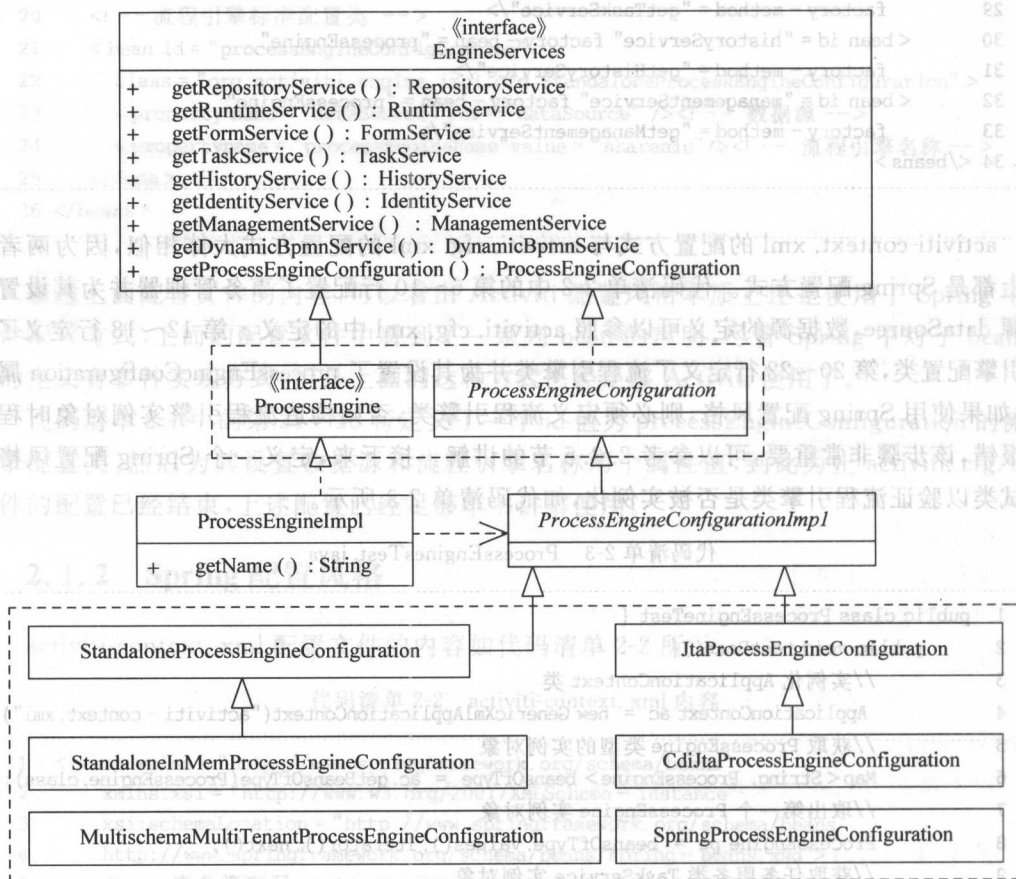


图 2-1 `ProcessEngineConfiguration` 子类边界

通过图 2-2 可以很清晰地从全局角度了解 `ProcessEngineConfiguration` 类,如果开发人员对 Activiti 源码没有太多接触,可能对图 2-2 中所涉及的类不是很理解,下面先简单介绍类的职责,后续章节会深入讲解。

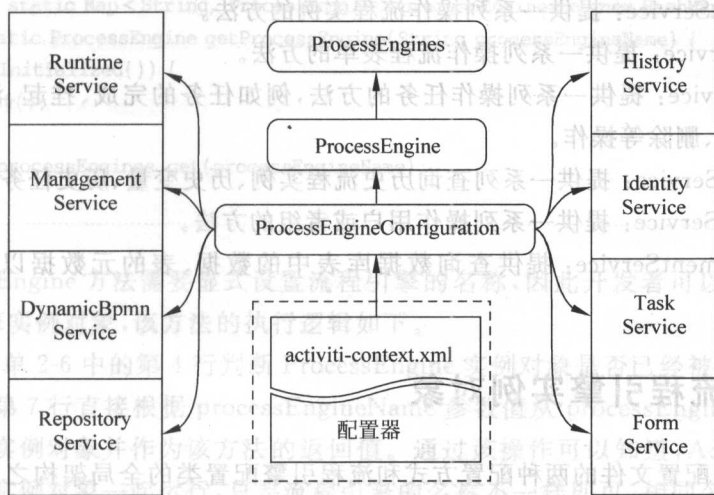


图 2-2 流程引擎架构图

- (1) EngineServices: 该接口中定义了获取各种服务类实例对象的方法。
- (2) ProcessEngine: 继承 EngineServices 接口,并增加了对流程引擎名称的获取以及关闭流程引擎的支持。
- (3) ProcessEngineImpl: 对 ProcessEngine 接口中定义的方法进行实现。
- (4) ProcessEngines: 该类负责管理所有的流程引擎 ProcessEngine 集合,并负责流程引擎实例对象的注册、获取、注销等操作。
- (5) ProcessEngineConfiguration: 该抽象类实现 EngineServices 接口,提供了一系列创建流程引擎配置类 ProcessEngineConfiguration 实例对象的方法。
- (6) ProcessEngineConfigurationImpl: 该抽象类继承 ProcessEngineConfiguration,负责创建一系列服务类实例对象、流程引擎实例对象以及 ProcessEngineImpl 类实例对象。该类可以通过流程配置文件交给 Spring 容器管理或者使用编程方式动态构造。
- (7) SpringProcessEngineConfiguration: 主要用于整合 Spring 框架时使用,提供几个重要功能:创建流程引擎实例对象,流程引擎启动之后自动部署配置的流程文档(需要设置),设置流程引擎连接的数据源、事务管理器等。
- (8) StandaloneProcessEngineConfiguration: 标准的流程引擎配置类。
- (9) MultiSchemaMultiTenantProcessEngineConfiguration: “多数据库多租户”流程引擎配置类,Activiti 通过此类为开发人员提供了自动路由机制,这样当流程引擎需要连接多个数据库进行操作时,客户端无须关心引擎到底连接的是哪一个数据库,该类通过路由规则自动选择需要操作的数据库,数据库的操作对客户端来说是透明的,客户端无须关心其内部路由实现机制。
- (10) JtaProcessEngineConfiguration: 顾名思义,通过类名也知道该类支持 JTA。
- (11) StandaloneInMemProcessEngineConfiguration: 该类通常可以在开发环境中自测使用,默认采用 H2 数据库存储数据。
- (12) EngineServices 提供的服务类如下。
 - RepositoryService: 提供一系列操作流程定义的方法。

- **RuntimeService**: 提供一系列操作流程实例的方法。
- **FormService**: 提供一系列操作流程表单的方法。
- **TaskService**: 提供一系列操作任务的方法, 例如任务的完成、挂起、激活、添加处理人、认领、删除等操作。
- **HistoryService**: 提供一系列查询历史流程实例、历史变量、历史任务的方法。
- **IdentityService**: 提供一系列操作用户或者组的方法。
- **ManagementService**: 提供查询数据库表中的数据、表的元数据以及执行命令等方法。

2.3 构造流程引擎实例对象

了解了流程配置文件的两种配置方式和流程引擎配置类的全局架构之后, 接下来开始讲解如何构造流程引擎实例对象, 首先使用 **ProcessEngines** 类创建 **ProcessEngine** 实例对象, 具体实现如代码清单 2-4 所示, 然后将上述两个流程配置文件放置到项目的 classpath 根路径中。

代码清单 2-4 ProcessEngineTest.java

```
1 @Test
2 public void getDefaultProcessEngine(){
3     //也可以为 getDefaultProcessEngine 方法传递参数, 参数的含义: 流程引擎的名称, 默认为 default
4     ProcessEngine pe = ProcessEngines.getDefaultProcessEngine();
5     TaskService taskService = pe.getTaskService();
6     assertNotNull("not null", taskService);
7 }
```

以上构造 **ProcessEngine** 实例对象的过程均是围绕 **ProcessEngines.getDefaultProcessEngine()** 这行代码展开, **getDefaultProcessEngine** 方法的相关实现, 如代码清单 2-5 所示。

代码清单 2-5 ProcessEngines.java

```
1 public static final String NAME_DEFAULT = "default"; //流程引擎的名称
2 public static ProcessEngine getDefaultProcessEngine() {
3     return getProcessEngine(NAME_DEFAULT);
4 }
```

getDefaultProcessEngine 方法仅仅是调用 **getProcessEngine** 方法进行下一步的处理, 并传入 **getProcessEngine** 方法需要的参数值 **default** (流程引擎的名称), 接下来分析 **getProcessEngine** 方法的处理逻辑, 该方法的相关实现如代码清单 2-6 所示。

代码清单 2-6 ProcessEngines.java

```
1 protected static boolean isInitialized = false;
```

```

2  protected static Map<String, ProcessEngine> processEngines = new HashMap();
3  public static ProcessEngine getProcessEngine(String processEngineName) {
4      if (!isInitialized()) {
5          init();
6      }
7      return processEngines.get(processEngineName);
8  }

```

getProcessEngine 方法需要显式设置流程引擎的名称,因此开发者可以直接调用该方法构造流程引擎实例对象,该方法的执行逻辑如下。

(1) 代码清单 2-6 中的第 4 行判断 ProcessEngine 实例对象是否已经被初始化,如果已经被初始化,则第 7 行直接根据 processEngineName 参数值从 processEngines 集合中查找 ProcessEngine 实例对象并作为该方法的返回值。通过该操作可以知道,Activiti 支持多个 ProcessEngine 实例对象一起运行,只要流程引擎的名称不一样即可,相同名称的流程引擎只会存在一个流程引擎实例,因为 processEngines 为 Map 数据结构,流程引擎名称的配置可以参考代码清单 2-1。流程引擎名称的默认值为 default。

(2) 如果 ProcessEngine 实例对象没有被初始化则调用 init 方法初始化 ProcessEngine 类,该方法的详细处理过程如代码清单 2-7 所示。

代码清单 2-7 ProcessEngines.java

```

1  public synchronized static void init() {
2      //再次确认流程引擎是否被初始化
3      if (!isInitialized()) {
4          if(processEngines == null) {
5              processEngines = new HashMap<String, ProcessEngine>();
6          }
7          ClassLoader classLoader = ReflectUtil.getClassLoader(); //获取类加载器
8          Enumeration<URL> resources = null;
9          try {
10             resources = classLoader.getResources("activiti.cfg.xml");
11             //定位 activiti.cfg.xml 文件
12         } catch (IOException e) {
13             throw new ActivitiIllegalArgumentException("retrieving activiti.cfg.xml, e);
14         }
15         Set<URL> configUrls = new HashSet<URL>();
16         while (resources.hasMoreElements()) {
17             configUrls.add(resources.nextElement()); //添加到 configUrls 集合
18         }
19         for (Iterator<URL> iterator = configUrls.iterator(); iterator.hasNext();) {
20             URL resource = iterator.next(); //循环遍历
21             initProcessEnginFromResource(resource);
22         }
23         try {
24             resources = classLoader.getResources("activiti-context.xml");
25         } catch (IOException e) {
26             throw new ActivitiIllegalArgumentException("retrieving activiti-context.xml, e);

```

```

27 while (resources.hasMoreElements()) {
28     URL resource = resources.nextElement();
29     initProcessEngineFromSpringResource(resource); //初始化流程引擎
30 }
31 setInitialized(true);
32 } else { }
33 }

```

下面根据代码清单 2-7 概括 init 方法的处理逻辑。

(1) 第 3 行再次确认 ProcessEngine 实例对象是否已经初始化, 经过确认该实例对象没有被初始化, 则开始执行后续操作, 否则不予处理。

(2) 如果 processEngines 集合为空, 则执行第 5 行初始化该集合, 该集合为 Map 数据结构, key 为 String 类型, 用于存储流程引擎的名称, value 值为 ProcessEngine 实例对象, 通过该集合的数据结构可知, 如果流程引擎的名称相同则只会存储一份 ProcessEngine 实例对象。

(3) 第 10 行定位 activiti.cfg.xml 文件的位置, 然后构造 ProcessEngine 实例对象。文件定位的工作比较简单, 首先获取类加载器 classLoader 对象, 然后委托 classLoader.getResources("activiti.cfg.xml") 方法定位流程配置文件, 通过该方法的处理逻辑可知该文件必须位于项目的根目录 classpath 中, 第 20 行调用 initProcessEngineFromResource 方法构造 ProcessEngine 实例对象。

(4) 第 23 行根据 classLoader 定位 activiti-context.xml 文件的位置, 然后开始构造 ProcessEngine 实例对象。如果该文件存在, 第 29 行直接委托 initProcessEngineFromSpringResource 方法构造 ProcessEngine 实例对象。

(5) 以上操作执行完毕后第 31 行调用 setInitialized(true) 方法, 如果该方法执行则表明构造 ProcessEngine 实例对象工作已经完毕。

上述为流程引擎初始化全过程概览, 接下来细化讲解该过程中涉及的内容, 首先讲解 Activiti 风格配置的流程引擎初始化过程。

2.3.1 初始化流程引擎之 Activiti 配置风格

initProcessEngineFromResource(resource) 方法负责解析 activiti.cfg.xml 文件中配置的 bean 信息, 该处理过程如代码清单 2-8 所示。

代码清单 2-8 ProcessEngines.java

```

1 private static ProcessEngineInfo initProcessEngineFromResource(URL resourceUrl) {
2     ProcessEngineInfo processEngineInfo = processEngineInfosByResourceUrl.get(resourceUrl.
    toString());
3     if (processEngineInfo != null) {
4         processEngineInfos.remove(processEngineInfo);
5         if (processEngineInfo.getException() == null) {
6             String processEngineName = processEngineInfo.getName();
7             processEngines.remove(processEngineName);
8             processEngineInfosByName.remove(processEngineName);

```

```

9     }
10    processEngineInfosByResourceUrl.remove(processEngineInfo.getResourceUrl());
11  }
12  String resourceUrlString = resourceUrl.toString();
13  try {
14      ProcessEngine processEngine = buildProcessEngine(resourceUrl);
15      String processEngineName = processEngine.getName();
16      processEngineInfo = new ProcessEngineInfoImpl (processEngineName, resourceUrlString,
17      null);
18      processEngines.put(processEngineName, processEngine);
19      processEngineInfosByName.put(processEngineName, processEngineInfo);
20  } catch (Throwable e) {
21      processEngineInfo = new ProcessEngineInfoImpl (null, resourceUrlString, getExceptionString
22      (e));
23      processEngineInfosByResourceUrl.put(resourceUrlString, processEngineInfo);
24      processEngineInfos.add(processEngineInfo);
25      return processEngineInfo;
26  }

```

进入 `initProcessEngineFromResource` 方法之后, Activiti 并不着急构造 `ProcessEngine` 实例对象,而是做了大量的准备工作,根据代码清单 2-8,其处理流程总结如下。

(1) 第 2 行根据 `resourceUrl` 参数值从 `processEngineInfosByResourceUrl` 集合中获取元素值,如果获取到,则第 4 行从 `processEngineInfos` 集合中移除该元素,第 5 行判断 `processEngineInfo` 对象是否存在异常信息,如果不存在异常信息,第 7 行从 `processEngines` 集合中移除该元素,紧接着第 8 行从 `processEngineInfosByName` 集合中移除该元素,最后不论 `processEngineInfo` 对象有无异常信息,都会执行第 10 行代码,从 `processEngineInfosByResourceUrl` 集合中移除该元素。

(2) 第 14 行将构造 `ProcessEngine` 实例对象的工作交给 `buildProcessEngine` 方法完成。

(3) 第 15~23 行开始向集合中添加元素,首先获取流程引擎的名称并实例化 `ProcessEngineInfoImpl` 类,然后分别对集合 `processEngineInfosByName`、`processEngineInfosByResourceUrl`、`processEngineInfos` 执行添加操作。

2.3.2 构造流程引擎实例对象

上面提到了调用 `buildProcessEngine` 方法构造 `ProcessEngine` 实例对象,接下来详细分析该方法的处理过程,如代码清单 2-9 所示。

代码清单 2-9 ProcessEngines.java

```

1 private static ProcessEngine buildProcessEngine(URL resource) {
2     InputStream inputStream = null;
3     try {
4         inputStream = resource.openStream();
5         ProcessEngineConfiguration processEngineConfiguration =

```



```

6     ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(inputStream);
7     return processEngineConfiguration.buildProcessEngine();
8 } catch (IOException e) {
9     throw new ActivitiIllegalArgumentException("couldn't open resource stream:", e);
10 } finally {
11     IoUtil.closeSilently(inputStream);
12 }
13 }

```

在 `buildProcessEngine(URL resource)` 方法中代码清单 2-9 中的第 4 行代码打开流程配置文件的数据流,第 5、第 6 行开始创建 `processEngineConfiguration` 实例对象,第 7 行调用 `processEngineConfiguration` 对象的 `buildProcessEngine` 方法构造 `ProcessEngine` 实例对象,最后执行第 11 行代码关闭文件流,虽然很简单却回收了资源。接下来重点分析第 5 行和第 6 行代码的实现逻辑。

2.3.3 创建流程引擎配置类实例

`ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(inputStream)` 方法完成流程引擎配置类 `ProcessEngineConfiguration` 的实例化工作,详细过程如代码清单 2-10 所示。

代码清单 2-10 `ProcessEngineConfiguration.java`

```

1 public static ProcessEngineConfiguration createProcessEngineConfigurationFromInputStream
  (InputStream inputStream) {
2     return createProcessEngineConfigurationFromInputStream ( inputStream, " processEngine-
  Configuration");
3 }
4 public static ProcessEngineConfiguration createProcessEngineConfigurationFromInputStream
  (InputStream inputStream, String beanName){
5     return BeansConfigurationHelper.parseProcessEngineConfigurationFromInputStream ( inputStream,
  beanName);
6 }

```

上述代码中,第 2 行直接调用第 4 行定义的方法进行处理,并传入 `beanName` 参数值 `processEngineConfiguration`,该参数值正是代码清单 2-1 中定义的 bean 的 id 值,第 4 行定义的方法处理逻辑也非常简单,直接委托 `BeansConfigurationHelper` 类中的 `parseProcessEngineConfigurationFromInputStream(inputStream, beanName)` 方法进行处理,该类的核心定义如代码清单 2-11 所示。

代码清单 2-11 `BeansConfigurationHelper.java`

```

1 public static ProcessEngineConfiguration parseProcessEngineConfiguration ( Resource
  springResource, String beanName) {
2     //实例化 Spring 框架中的 DefaultListableBeanFactory 类
3     DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();
4     XmlBeanDefinitionReader xmlBeanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

```

```
5 //设置验证模式为 XSD,当然也支持 DTD 方式验证
6 xmlBeanDefinitionReader.setValidationMode(XmlBeanDefinitionReader.VALIDATION_XSD);
7 //加载读取到的 springResource 资源并交给 Spring 进行解析
8 xmlBeanDefinitionReader.loadBeanDefinitions(springResource);
9 //通过 beanFactory 对象获取 ProcessEngineConfigurationImpl 实例对象
10 ProcessEngineConfigurationImpl processEngineConfiguration = (ProcessEngineConfigurationImpl)
    beanFactory.getBean(beanName);
11 //将 beanFactory 对象使用 SpringBeanFactoryProxyMap 进行包装
12 processEngineConfiguration.setBeans(new SpringBeanFactoryProxyMap(beanFactory));
13 return processEngineConfiguration;
14 }
15 public static ProcessEngineConfiguration parseProcessEngineConfigurationFromInputStream
    (InputStream inputStream, String beanName) {
16 //将 inputStream 对象转化为 Spring 中的 Resource 对象
17 Resource springResource = new InputStreamResource(inputStream);
18 //实例化流程配置文件中的 bean
19 return parseProcessEngineConfiguration(springResource, beanName);
20 }
21 public static ProcessEngineConfiguration parseProcessEngineConfigurationFromResource
    (String resource, String beanName) {
22 Resource springResource = new ClassPathResource(resource);
23 return parseProcessEngineConfiguration(springResource, beanName);
24 }
```

通过分析 BeansConfigurationHelper 类中定义的三个方法可以得出,Activiti 底层通过该类巧妙地将流程配置文件中定义的 bean 全部交给 Spring 容器进行管理(包括类的配置、加载和获取操作),结合图 2-3 总结 parseProcessEngineConfiguration 方法的执行逻辑。

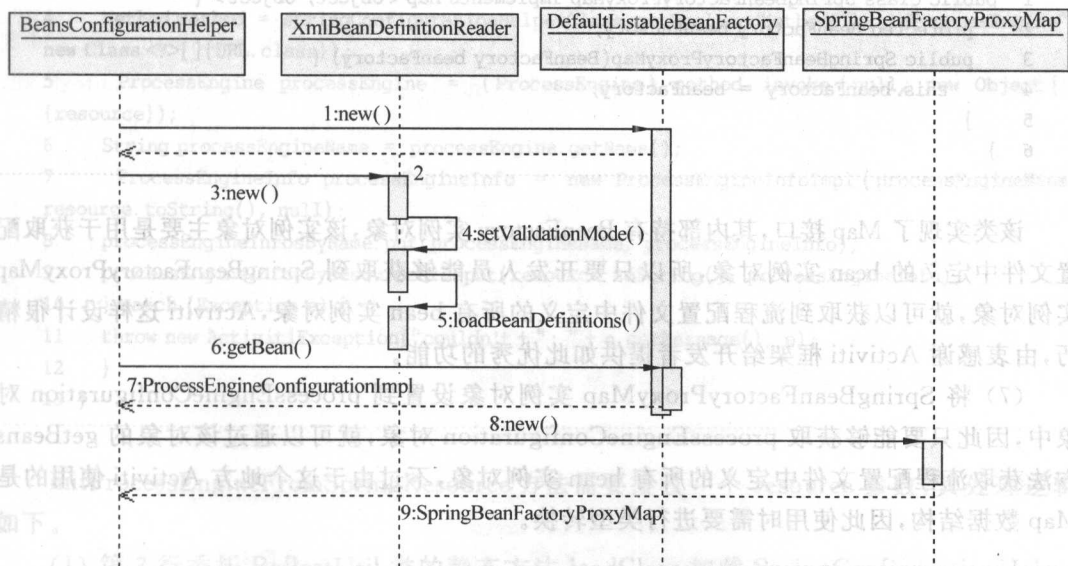


图 2-3 Activiti 类的配置、加载和获取操作

(1) 第 3 行实例化 DefaultListableBeanFactory 类。

(2) 第 4 行实例化 XmlBeanDefinitionReader 类, 并将 beanFactory 设置到 xmlBeanDefinitionReader 对象中, 该类非常重要, Spring 框架使用 XmlBeanDefinitionReader 类读取和解析 XML 文件。

(3) 第 6 行设置 xmlBeanDefinitionReader 对象的验证模式为 XSD。

(4) 第 8 行 xmlBeanDefinitionReader.loadBeanDefinitions(springResource) 方法加载配置文件 activiti.cfg.xml 中定义的所有 bean。

(5) 第 10 行使用 beanFactory 对象的 getBean 方法获取 ProcessEngineConfigurationImpl 实例对象, 对应代码清单 2-1 配置的 StandaloneProcessEngineConfiguration 类, 这就是最典型的 Spring 获取 bean 实例对象的操作方式。实例化 StandaloneProcessEngineConfiguration 类的同时会初始化其父类 ProcessEngineConfigurationImpl 中的各种属性值, 如代码清单 2-12 所示。

代码清单 2-12 ProcessEngineConfigurationImpl.java

```
1 protected RepositoryService repositoryService = new RepositoryServiceImpl();
2 protected RuntimeService runtimeService = new RuntimeServiceImpl();
3 ...//各种服务类
```

以上这些服务类在实际项目开发中经常使用到, 但通常是根据 EngineServices 实例对象获取以上这些服务类的实例对象, 那么为什么这些服务类的实例化工作在 ProcessEngineConfigurationImpl 类中进行呢? 暂且留下一个悬念, 稍后讲解。

(6) 实例化 SpringBeanFactoryProxyMap 类, 该类的定义如代码清单 2-13 所示。

代码清单 2-13 SpringBeanFactoryProxyMap.java

```
1 public class SpringBeanFactoryProxyMap implements Map< Object, Object> {
2     protected BeanFactory beanFactory;
3     public SpringBeanFactoryProxyMap(BeanFactory beanFactory) {
4         this.beanFactory = beanFactory;
5     }
6 }
```

该类实现了 Map 接口, 其内部持有 BeanFactory 实例对象, 该实例对象主要是用于获取配置文件中定义的 bean 实例对象, 所以只要开发人员能够获取到 SpringBeanFactoryProxyMap 实例对象, 就可以获取到流程配置文件中定义的所有 bean 实例对象, Activiti 这样设计很精巧, 由衷感谢 Activiti 框架给开发者提供如此优秀的功能。

(7) 将 SpringBeanFactoryProxyMap 实例对象设置到 processEngineConfiguration 对象中, 因此只要能够获取 processEngineConfiguration 对象, 就可以通过该对象的 getBeans 方法获取流程配置文件中定义的所有 bean 实例对象, 不过由于这个地方 Activiti 使用的是 Map 数据结构, 因此使用时需要进行类型转换。

2.3.4 初始化流程引擎

代码清单 2-9 中, buildProcessEngine() 方法用于创建 ProcessEngine 实例对象, 因为

activiti.cfg.xml 配置文件中定义的流程引擎配置类为 StandaloneProcessEngineConfiguration, 但是该类中并没有定义 buildProcessEngine 方法, 那么 buildProcessEngine 方法肯定在其父类中进行了实现, 接下来分析该方法在其父类中的处理逻辑, 如代码清单 2-14 所示。

代码清单 2-14 ProcessEngineConfigurationImpl.java

```
1 public ProcessEngine buildProcessEngine() {  
2     init();  
3     return new ProcessEngineImpl(this);  
4 }
```

该方法的处理逻辑总结如下。

- (1) 调用 init 方法初始化各种属性值。
- (2) 实例化 ProcessEngineImpl 类。

2.3.5 初始化流程引擎之 Spring 配置风格

上述一系列的讲解均是针对 Activiti 配置文件的风格构造 ProcessEngine 实例对象, 接下来详细分析 Spring 风格的配置文件 activiti-context.xml (可以参考 2.1.2 节) 构造 ProcessEngine 实例对象的整个过程, 以下讲解均是围绕代码清单 2-7 中的 initProcessEngineFromSpringResource(resource) 方法展开的, 该方法的实现如代码清单 2-15 所示。

代码清单 2-15 ProcessEngines.java

```
1 protected static void initProcessEngineFromSpringResource(URL resource) {  
2     try {  
3         Class springConfigurationHelperClass = ReflectUtil.loadClass("org.activiti.spring.  
SpringConfigurationHelper");  
4         Method method = springConfigurationHelperClass.getDeclaredMethod("buildProcessEngine",  
new Class[] { URL.class });  
5         ProcessEngine processEngine = (ProcessEngine) method.invoke(null, new Object[]  
{ resource });  
6         String processEngineName = processEngine.getName();  
7         ProcessEngineInfo processEngineInfo = new ProcessEngineInfoImpl ( processEngineName,  
resource.toString(), null );  
8         processEngineInfosByName.put(processEngineName, processEngineInfo);  
9         processEngineInfosByResourceUrl.put(resource.toString(), processEngineInfo);  
10    } catch (Exception e) {  
11        throw new ActivitiException("couldn't + ": " + e.getMessage(), e);  
12    }  
13 }
```

initProcessEngineFromSpringResource 方法需要接收一个 resource 参数, 其处理逻辑如下。

- (1) 第 3 行委托 ReflectUtil 类的静态方法 loadClass 加载 SpringConfigurationHelper 类(该类位于 activiti-spring-5.21.0.jar 包中)。关于类加载机制本书不详细讲解, 其相关实现可跟进 loadClass 方法进行查看。

(2) 第 4 行通过 `springConfigurationHelperClass` 对象查找 `SpringConfigurationHelper` 类中的 `buildProcessEngine` 方法。

(3) 第 5 行通过反射方式调用 `SpringConfigurationHelper` 类中的 `buildProcessEngine` 方法,然后使用 `processEngine` 变量存储该方法的返回值。

(4) 第 6 行获取流程引擎的名称。

(5) 以上步骤执行完后,第 7~9 行将流程引擎的详细信息添加到 `processEngineInfosByName` 和 `processEngineInfosByResourceUrl` 集合中。

2.3.5.1 反射构造 ProcessEngine

上面代码通过反射方式调用 `SpringConfigurationHelper` 类中的 `buildProcessEngine` 方法,该方法的定义如代码清单 2-16 所示。

代码清单 2-16 `SpringConfigurationHelper.java`

```
1 public static ProcessEngine buildProcessEngine(URL resource) {
2     ApplicationContext applicationContext = new GenericXmlApplicationContext(new UriResource
(resource));
3     Map beansOfType = applicationContext.getBeansOfType(ProcessEngine.class);
4     if ((beansOfType == null) || (beansOfType.isEmpty())) {
5         throw new ActivitiException("no " + resource.toString());
6     }
7     ProcessEngine processEngine = beansOfType.values().iterator().next();
8     return processEngine;
9 }
```

该方法的处理逻辑总结如下。

(1) 第 2 行加载 `activiti-context.xml` 文件。

(2) 第 3 行获取类型为 `ProcessEngine` 的实例对象,第 4~5 行如果 `activiti-context.xml` 文件中没有定义类型为 `ProcessEngine` 的 bean 则程序报错,第 7 行如果发现 `activiti-context.xml` 文件中配置了多个类型为 `ProcessEngine` 的 bean,则从 `beansOfType` 集合中取出第一个 `ProcessEngine` 实例对象并作为方法的返回值,从 `beansOfType` 集合的处理逻辑可以看出流程配置文件中不管定义多少个 `ProcessEngine` 类,程序只会使用一个而且是第一个。接下来重点分析 `activiti-context.xml` 文件中定义的 `ProcessEngineFactoryBean` 类是如何被 Spring 实例化的,首先查看代码清单 2-2 中定义的 `ProcessEngineFactoryBean`,该工厂类负责生成 `ProcessEngine` 实例对象,暂且将关注点放到 `ProcessEngineFactoryBean` 类的 `getObject` 方法的处理逻辑中,如代码清单 2-17 所示。

代码清单 2-17 `ProcessEngineFactoryBean.java`

```
1 public class ProcessEngineFactoryBean implements FactoryBean<ProcessEngine>
2     public ProcessEngine getObject() throws Exception {
3         configureExpressionManager(); //设置表达式管理器
4         configureExternallyManagedTransactions(); //设置事务管理器
5         if(processEngineConfiguration.getBeans() == null) { //设置 bean
6             processEngineConfiguration.setBeans(new SpringBeanFactoryProxyMap(applicationContext));
```



```

7     }
8     //开始构造 ProcessEngine 的实例对象
9     this.processEngine = processEngineConfiguration.buildProcessEngine();
10    return this.processEngine;
11}

```

该方法的处理逻辑总结如下。

- (1) 第 3 行调用 configureExpressionManager 方法设置表达式管理器。
- (2) 第 4 行调用 configureExternallyManagedTransactions 方法设置事务管理器, 具体实现如代码清单 2-18 所示。

代码清单 2-18 ProcessEngineFactoryBean.java

```

1  protected void configureExternallyManagedTransactions() {
2      if (processEngineConfiguration instanceof SpringProcessEngineConfiguration) {
3          SpringProcessEngineConfiguration engineConfiguration = (SpringProcessEngineConfiguration)
processEngineConfiguration;
4          if (engineConfiguration.getTransactionManager() != null) {
5              processEngineConfiguration.setTransactionsExternallyManaged(true);
6          }
7      }
8  }

```

如果 processEngineConfiguration 对象为 SpringProcessEngineConfiguration 实例对象, 则进行如下处理, 否则不予理会。

- 将 processEngineConfiguration 对象转换为 SpringProcessEngineConfiguration 实例对象。
- 如果 engineConfiguration 对象中的 getTransactionManager 方法返回值不为空, 则将事务交给 Spring 框架管理, 可以参考 12.8 节。通过这里的处理可以发现, 如果使用 Spring 管理事务, 则流程引擎配置类必须为 SpringProcessEngineConfiguration。

(3) 第 9 行开始构造流程引擎实例对象, processEngineConfiguration 为 SpringProcessEngineConfiguration 类型。buildProcessEngine 方法的处理逻辑如代码清单 2-19 所示。

代码清单 2-19 SpringProcessEngineConfiguration.java

```

1  public ProcessEngine buildProcessEngine() {
2      ProcessEngine processEngine = super.buildProcessEngine();
3      ProcessEngines.setInitialized(true);
4      autoDeployResources(processEngine);
5      return processEngine;
6  }

```

buildProcessEngine 方法的处理逻辑总结如下。

- 第 2 行委托父类构造 ProcessEngine 实例对象, 因为当前类 SpringProcessEngineConfiguration 的父类为 ProcessEngineConfigurationImpl, 所以该操作会触发代码

清单 2-14 中的逻辑。

- 第 3 行通知 ProcessEngines 类, ProcessEngine 实例已经初始化。
- 第 4 行开始自动部署流程资源。

Activiti 整合 Spring 框架时, 提供了自动部署资源的特性, 这样流程引擎启动时会自动将指定的资源文件部署到数据库中, 具体的使用方式可以参照代码清单 2-2 中的配置, autoDeployResources(processEngine) 方法的处理逻辑比较简单, 本书不做过多解释。

注意

ProcessEngineFactoryBean 类实现了 FactoryBean 接口。

2.4 初始化流程引擎配置类

在代码清单 2-14 中, buildProcessEngine 方法会调用 init() 方法初始化 ProcessEngine-ConfigurationImpl 实例对象的各种属性, 具体实现如代码清单 2-20 所示。

代码清单 2-20 ProcessEngineConfigurationImpl.java

```

1  protected void init() {
2      initConfigurators();           //初始化配置器
3      configuratorsBeforeInit();     //调用配置器的 beforeInit 方法
4      initProcessDiagramGenerator(); //初始化流程图片生成器
5      initHistoryLevel();            //初始化历史记录归档级别, 默认为 AUDIT 级别
6      initExpressionManager();       //初始化表达式管理器
7      initDataSource();              //初始化数据源
8      initVariableTypes();           //初始化变量类型
9      initBeans();                   //初始化可以管理的 bean
10     initFormEngines();              //初始化表单引擎
11     initFormTypes();               //初始化表单类型
12     initScriptingEngines();        //初始化脚本引擎
13     initClock();                   //初始化时间类, 主要负责提供设置当前时间等
14     initBusinessCalendarManager(); //初始化日期管理器
15     initCommandContextFactory();    //初始化命令上下文工厂
16     initTransactionContextFactory(); //初始化事务上下文工厂
17     initCommandExecutors();         //初始化命令执行器
18     initServices();                 //为各种服务类对象, 比如 repositoryService 设置命令执行器
19     initIdGenerator();              //初始化 id 生成器
20     initDeployers();                //初始化部署器
21     initJobHandlers();              //初始化定时处理类
22     initJobExecutor();              //初始化定时任务执行器
23     initAsyncExecutor();            //初始化异步执行器
24     initTransactionFactory();        //初始化事务工厂
25     initSqlSessionFactory();        //初始化 SqlSession 工厂
26     initSessionFactory();           //初始化 Session 工厂
27     initJpa();                      //初始化 JPA
28     initDelegateInterceptor();      //负责处理拦截器默认实现类(拦截监听器或者表达式)
29     initEventHandlers();            //初始化事件处理类
30     initFailedJobCommandFactory();  //初始化失败命令工厂

```

```

31  initEventDispatcher();           //初始化事件转发器
32  initProcessValidator();          //初始化流程验证器
33  initDatabaseEventLogging();      //初始化数据库事件记录
34  configuratorsAfterInit();        //调用配置器的 configure 方法
35  }

```

仅从代码量上就能看出 ProcessEngineConfigurationImpl 类的初始化相当复杂,涉及了各种各样的考虑。在细化讲解之前,首先要明白一点 ProcessEngineConfigurationImpl 是抽象类,上文讲解的两种配置风格中使用到的两个流程引擎配置类 StandaloneProcessEngineConfiguration 和 SpringProcessEngineConfiguration 均继承 ProcessEngineConfigurationImpl 类。

在实际项目开发中如果开发人员觉得 ProcessEngineConfigurationImpl 类中的初始化方法不能满足业务需求,例如 initBeans 方法不能满足要求,则可以自定义一个类继承 StandaloneProcessEngineConfiguration 或者 SpringProcessEngineConfiguration,然后重写 initBeans 方法。由于 init 方法中涉及了大量属性的初始化工作,如果单一地对每个方法进行讲解恐怕很难理解其精髓,而且可能会有事倍功半的效果,因此本章节先讲解几个比较重要的初始化方法,其余后续章节会陆续讲解,进而达到事半功倍的效果。

以上绝大部分方法的初始化处理逻辑均为:首先判断客户端是否设置了指定的属性值,如果客户端设置了则优先使用,否则使用系统内置的值进行初始化工作,对于这些可以让客户端扩展的属性,将其称之为“开关属性”也许会更加容易理解一点。

约定

本书中如果没有特殊说明,则开关属性均指 ProcessEngineConfigurationImpl 类中可以让客户端扩展的属性。

2.5 配置器

代码清单 2-20 中 initConfigurators 方法所做的工作就是初始化“配置器”,上文讲解的两种配置风格的配置文件中分别定义了不同的流程引擎配置类,例如 StandaloneProcessEngineConfiguration 类,但是在 XML 中定义流程引擎配置类有如下三个缺点。

- (1) 不灵活,所有的属性信息都需要在 XML 文档中进行配置。
- (2) 不能满足动态属性配置需求,如果开发人员打算使用编程方式构造流程引擎配置类的实例对象,则这种方式几乎不可能实现。
- (3) 不能检查必要的属性值是否已经被初始化,比如开发人员期望检查数据源的信息,则配置方式不能达到这个诉求。

2.5.1 初始化配置器

Activiti 在 5.13 版本中增加了“配置器”,进而可以通过编程的方式动态修改或者刷新流程引擎配置类实例对象中的属性值,所有的配置器均需要实现 ProcessEngineConfigurator 接口。下面分析 initConfigurators 方法的处理过程,如代码清单 2-21 所示。

代码清单 2-21 ProcessEngineConfigurationImpl.java

```

1  protected boolean enableConfiguratorServiceLoader = true;
2  protected List<ProcessEngineConfigurator> allConfigurators;
3  protected List<ProcessEngineConfigurator> configurators;
4  protected void initConfigurators() {
5      allConfigurators = new ArrayList<ProcessEngineConfigurator>();
6      if (configurators != null) {
7          for (ProcessEngineConfigurator configurator : configurators) {
8              allConfigurators.add(configurator);
9          }
10     }
11     if (enableConfiguratorServiceLoader) {
12         ClassLoader classLoader = getClassLoader();
13         if (classLoader == null) {
14             classLoader = ReflectUtil.getClassLoader();
15         }
16         ServiceLoader<ProcessEngineConfigurator> configuratorServiceLoader
17             = ServiceLoader.load(ProcessEngineConfigurator.class, classLoader);
18         for (ProcessEngineConfigurator configurator : configuratorServiceLoader) {
19             allConfigurators.add(configurator);
20         }
21         if (!allConfigurators.isEmpty()) {
22             Collections.sort(allConfigurators, new Comparator<ProcessEngineConfigurator>() {
23                 public int compare(ProcessEngineConfigurator c1, ProcessEngineConfigurator c2) {
24                     int priority1 = c1.getPriority();
25                     int priority2 = c2.getPriority();
26                     if (priority1 < priority2) {
27                         return -1;
28                     } else if (priority1 > priority2) {
29                         return 1;
30                     }
31                     return 0;
32                 }
33             });
34         }
35     }

```

下面对 initConfigurators 方法的处理逻辑加以总结。

- (1) 第 5 行实例化 allConfigurators 集合, 该集合用于存储所有的配置器实例。
- (2) 第 6 行判断开关属性 configurators 是否为空, 如果该属性值不为空, 则循环遍历该集合, 并将遍历的值添加到 allConfigurators 集合中。
- (3) 判断开关属性 enableConfiguratorServiceLoader 是否为 true, 如果是则执行后续操作, 该值默认为 true。

- 第 12 行获取类加载器实例 classLoader。

- 第 16~17 行利用 Java 中的 ServiceLoader 特性加载 ProcessEngineConfigurator 实例集合, 第 18~19 行循环遍历 configuratorServiceLoader 集合并将遍历的值添加到 allConfigurators 集合中。

第 21 行判断 allConfigurators 集合是否为空,如果不为空,则执行第 22~30 行对该集合中的元素按照 ProcessEngineConfigurator 类中 getPriority 方法的返回值进行升序排序。

注意

classLoader 为开关属性。

2.5.2 配置器实战

上面详细讲解了配置器的初始化过程,下面讨论一下配置器的全局架构(如图 2-4 所示)。

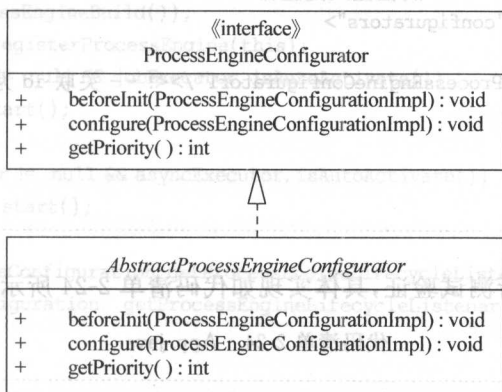


图 2-4 ProcessEngineConfigurator 架构图

(1) ProcessEngineConfigurator: 该接口定义了 beforeInit 方法(在 configuratorsBeforeInit 方法中调用)、configure 方法(在 configuratorsAfterInit 方法中调用)、getPriority 方法(在初始化配置器方法 initConfigurators 中调用)。

(2) AbstractProcessEngineConfigurator: 对 ProcessEngineConfigurator 接口进行了实现,作为模板抽象类存在,并重写了 getPriority 方法,该方法的返回值默认为 10 000。

了解以上知识点之后,接下来学习如何自定义配置器并修改流程引擎配置类实例的属性值,相关实现如代码清单 2-22 所示。

代码清单 2-22 ShareniuConfiguratorA.java

```
1 public class ShareniuConfiguratorA extends AbstractProcessEngineConfigurator {
2     public int getPriority() {
3         return 1;
4     }
5     public void beforeInit(ProcessEngineConfigurationImpl processEngineConfiguration) {
6         System.out.println("A:beforeInit");
7         processEngineConfiguration.setDatabaseSchemaUpdate("true"); //设置属性值
8     }
9     public void configure(ProcessEngineConfigurationImpl processEngineConfiguration) {
10        System.out.println("A:configure");
11    }
12 }
```


定义了配置器之后,接下来的工作就是把自定义配置器注入流程引擎配置类,如代码清单 2-23 所示。

代码清单 2-23 activiti.cfg.xml

```

1 < bean id = "myProcessEngineConfigurator1" class = "com.shareniu.chapter2.configurator.ShareniuConfiguratorA"></bean>
2 < bean id = "processEngineConfiguration"
3 class = "org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
4 < property name = "dataSource" ref = "dataSource"></property>
5 <!-- 关闭 ServiceLoader 功能 -->
6 < property name = "enableConfiguratorServiceLoader" value = "false"></property>
7 <!-- 通过 configurators 属性注入配置器 -->
8 < property name = "configurators">
9 <list>
10 <ref bean = "myProcessEngineConfigurator1"/><!-- 关联 id 为 shareniuConfiguratorA 的 bean -->
11 </list>
12 </property>
13 </bean>

```

对自定义配置器进行测试验证,具体实现如代码清单 2-24 所示。

代码清单 2-24 App.java

```

1 public void init() {
2     InputStream in =
3     App.class.getClassLoader().getResourceAsStream("com/shareniu/chapter2/configurator/activiti.cfg.xml");
4     ProcessEngineConfiguration pcf =
5     ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(in);
6     ProcessEngine processEngine = pcf.buildProcessEngine();
7 }

```

执行上述代码,得到输出结果如下:

```

A: beforeInit
A: configure

```

在学习自定义配置器的同时,也学会了如何通过获取以及设置 ProcessEngineConfigurationImpl 实例对象的属性值,关于配置器更高级的用法以及使用技巧可以参考 8.10.4 节。

2.6 初始化流程引擎

在代码清单 2-14 中, new ProcessEngineImpl(this) 这行代码主要用于实例化 ProcessEngineImpl 类,该类的核心定义如代码清单 2-25 所示。

代码清单 2-25 ProcessEngineImpl.java

```

1 public ProcessEngineImpl(ProcessEngineConfigurationImpl processEngineConfiguration) {

```

```

2  this.processEngineConfiguration = processEngineConfiguration; //流程引擎配置类实例
3  this.name = processEngineConfiguration.getProcessEngineName(); //流程引擎的名称
4  //初始化各种服务类实例
5  this.repositoryService = processEngineConfiguration.getRepositoryService();
6  .....//设置一系列服务类
7  this.jobExecutor = processEngineConfiguration.getJobExecutor(); //定时作业执行器
8  this.asyncExecutor = processEngineConfiguration.getAsyncExecutor(); //异步作业执行器
9  this.commandExecutor = processEngineConfiguration.getCommandExecutor(); //命令执行器
10 this.sessionFactories = processEngineConfiguration.getSessionFactories();
11 //事务上下文工厂类
12 this.transactionContextFactory = processEngineConfiguration.getTransactionContextFactory();
13 commandExecutor.execute ( processEngineConfiguration.getSchemaCommandConfig ( ), new
SchemaOperationsProcessEngineBuild());
14 ProcessEngines.registerProcessEngine(this);
15 if (jobExecutor != null && jobExecutor.isAutoActivate()) {
16     jobExecutor.start();
17 }
18 if (asyncExecutor != null && asyncExecutor.isAutoActivate()) {
19     asyncExecutor.start();
20 }
21 if (processEngineConfiguration.getProcessEngineLifecycleListener() != null) {
22     processEngineConfiguration.getProcessEngineLifecycleListener ( ). onProcessEngineBuilt
(this);
23 }
24 processEngineConfiguration.getEventDispatcher().dispatchEvent(
25     ActivitiEventBuilder.createGlobalEvent(ActivitiEventType.ENGINE_CREATED));
26 }

```

尝试将以上代码的执行逻辑梳理总结如下。

(1) 第 2 ~ 12 行进行属性填充, ProcessEngineImpl 实例对象的属性值均从 processEngineConfiguration 对象中进行获取, 这里的设计是典型的门面模式, 为客户端的使用提供便利, 无须关心 ProcessEngineConfiguration 类的内部实现机制。

(2) 第 13 行执行数据库表生成策略。开发人员可以在配置文件中为流程引擎配置类设置 databaseSchemaUpdate 属性值, 该属性的可选值如下所示。

- false: 默认值。流程引擎启动时, 首先从 ACT_GE_PROPERTY 表中查询 Activiti 引擎的版本值 (NAME_字段的值等于 'schema.version'), 然后获取 ProcessEngine 接口中定义的 VERSION 静态变量值, 两者进行对比, 如果数据库中的表不存在或者表存在但版本不匹配则直接抛出异常。
- true: 流程引擎启动时会对所有的表进行更新操作 (upgrade 目录中的 DDL 脚本), 如果数据库中的表不存在则开始创建表 (create 目录中的 DDL 脚本)。
- create_drop: 流程引擎启动时创建表, 流程引擎关闭时删除表 (流程引擎的关闭形如 processEngine.close())。
- drop-create: 流程引擎启动时首先删除数据库中存在的表然后重新创建表 (该方式不需要手动关闭流程引擎), 该操作非常危险, 因此不建议正式环境使用。
- create: 流程引擎启动时直接创建表不管数据库是否存在表, 这就意味着如果数据

库中已经存在表,再次执行创建表的 DDL 肯定会报错,因此不建议使用。

(3) 注册流程引擎。ProcessEngines.registerProcessEngine(this) 方法将 ProcessEngineImpl 实例对象注册到 ProcessEngines 类中。

(4) 作业执行器。第 15~20 行如果流程引擎配置类配置了作业执行器 jobExecutorActivate 开关属性和异步作业执行器 asyncExecutorActivate 开关属性则需要分别启动作业执行器和异步作业执行器。有关作业执行器可以参考第 9 章的讲解。

(5) 流程引擎生命周期监听器。如果流程引擎配置类配置有流程引擎生命周期监听器 ProcessEngineLifecycleListener,则执行第 22 行触发流程引擎生命周期监听器中的 onProcessEngineBuilt 方法。

(6) 转发事件。第 24~26 行转发事件类型为 ENGINE_CREATED 的事件,关于事件转发器可以参考第 6 章。

2.6.1 操作引擎表

了解 databaseSchemaUpdate 属性的含义之后,接下来分析 SchemaOperationsProcessEngineBuild 命令类中 execute 方法的处理逻辑,如代码清单 2-26 所示。有关命令类的相关知识,第 12 章会详细讲解。

代码清单 2-26 SchemaOperationsProcessEngineBuild.java

```
1 public Object execute(CommandContext commandContext) {
2     commandContext.getSession(DbSqlSession.class)
3     .performSchemaOperationsProcessEngineBuild();
4     return null;
5 }
```

首先根据 commandContext 获取 DbSqlSession 实例对象,然后调用该实例对象的 performSchemaOperationsProcessEngineBuild 方法进行下一步的处理,该方法的具体实现如代码清单 2-27 所示。

代码清单 2-27 DbSqlSession.java

```
1 public void performSchemaOperationsProcessEngineBuild() {
2     String databaseSchemaUpdate = Context.getProcessEngineConfiguration().getDatabaseSchemaUpdate();
3     if ("drop-create".equals(databaseSchemaUpdate)) {
4         try {
5             dbSchemaDrop();
6         } catch (RuntimeException e) {
7         }
8     }
9     if ("create-drop".equals(databaseSchemaUpdate)
10        || "drop-create".equals(databaseSchemaUpdate)
11        || "create".equals(databaseSchemaUpdate)
12    ) {
13        dbSchemaCreate();
14    }
```

```

14
15 } else if ("false".equals(databaseSchemaUpdate)) {
16     dbSchemaCheckVersion();
17
18 } else if ("true".equals(databaseSchemaUpdate)) {
19     dbSchemaUpdate();
20 }
21 }

```

该方法的处理逻辑梳理如下。

- (1) 根据 Context 类获取流程引擎配置类中的 databaseSchemaUpdate 属性值。
- (2) 根据 databaseSchemaUpdate 值执行如下的逻辑。
 - 如果属性值为 drop-create, 则第 5 行调用 dbSchemaDrop 方法进行数据库表的删除工作。
 - 如果属性值为 create-drop、drop-create、create 三者中的任意一个, 则第 13 行调用 dbSchemaCreate 方法进行数据库表的创建工作。
 - 如果该属性值为 false, 则第 16 行调用 dbSchemaCheckVersion 方法对数据库中存在的版本值与 ProcessEngine 接口中定义的版本值进行比对。
 - 如果该属性值为 true, 则第 19 行调用 dbSchemaUpdate 方法进行数据库表的升级操作。

以上数据库的 DDL 文件在 1.3 节详细讲解过, 可以参考该节进行学习。

扩展

如果打算在引擎创建、删除、更新表时执行项目中的 DDL 脚本, 可以通过扩展 DbSqlSession 类进行实现。

2.7 管理流程引擎

2.7.1 注册流程引擎

上文多次提到流程引擎对象构造完毕, 会将自身信息注册到流程引擎管理类中, 以方便后续操作, 这一过程从 ProcessEngines.registerProcessEngine(this) 开始, 该方法的详细实现如代码清单 2-28 所示。

代码清单 2-28 ProcessEngines.java

```

1 protected static Map<String, ProcessEngine> processEngines = new HashMap<>();
2 public static void registerProcessEngine(ProcessEngine processEngine) {
3     processEngines.put(processEngine.getName(), processEngine);
4 }

```

毕竟 registerProcessEngine 方法仅仅是注册流程引擎实例对象, 所以实现逻辑比较简单, 直接将 ProcessEngine 实例对象添加到 ProcessEngines 类中的 processEngines 集合即可。

2.7.2 关闭流程引擎

上文分析了流程引擎的注册过程,接下来详细分析流程引擎的注销过程,流程引擎关闭时会调用注销方法,上文提到过流程引擎的关闭操作形如 `processEngine.close()`, `close` 方法负责关闭流程引擎,该方法的详细实现如代码清单 2-29 所示。

代码清单 2-29 ProcessEngineImpl.java

```

1 public void close() {
2     ProcessEngines.unregister(this);
3     if (jobExecutor != null && jobExecutor.isActive()) {
4         jobExecutor.shutdown();
5     }
6     if (asyncExecutor != null && asyncExecutor.isActive()) {
7         asyncExecutor.shutdown();
8     }
9     commandExecutor.execute(processEngineConfiguration.getSchemaCommandConfig(), new
SchemaOperationProcessEngineClose());
10    if (processEngineConfiguration.getProcessEngineLifecycleListener() != null) {
11        processEngineConfiguration.getProcessEngineLifecycleListener().onProcessEngineClosed
(this);
12    }
13    processEngineConfiguration.getEventDispatcher().dispatchEvent(
14        ActivitiEventBuilder.createGlobalEvent(ActivitiEventType.ENGINE_CLOSED));
15 }

```

下面对 `close` 方法的执行逻辑加以总结。

(1) 第 2 行注销流程引擎实例。

(2) 关闭执行器。

第 3~7 行如果流程引擎配置类配置了作业执行器 `jobExecutorActivate` 开关属性和异步作业执行器 `asyncExecutorActivate` 开关属性,则需要分别关闭上述的两个作业执行器。

(3) 执行 `SchemaOperationProcessEngineClose` 命令。

(4) 流程引擎生命周期监听器。

如果流程引擎配置类配置了流程引擎生命周期监听器,则第 11 行触发流程引擎生命周期监听器中的 `onProcessEngineClosed` 方法。

(5) 转发事件。第 13~14 行转发 `ENGINE_CLOSED` 事件。

如果使用 `StandaloneProcessEngineConfiguration` 实例对象,则需要手动调用流程引擎的 `close` 方法,如果使用 `ProcessEngineFactoryBean` 类构造流程引擎,则无须关心 `close` 方法,具体实现逻辑可以跟进该类的 `destroy` 方法进行查看。

2.8 流程引擎生命周期监听器

上文讲解过流程引擎实例化和关闭时会触发 `ProcessEngineLifecycleListener` 类中的不同方法,分别对应 `onProcessEngineBuilt` 和 `onProcessEngineClosed`,开发人员可以很方便

地获取流程引擎的创建或关闭事件,从而达到监听流程引擎整个生命周期的目的,但是 Activiti 中并没有提供流程引擎生命周期监听器的默认实现类,因此本节重点讲解如何使用流程引擎生命周期监听器,首先定义一个类,具体实现如代码清单 2-30 所示。

代码清单 2-30 ShareniuLifecycleListener.java

```
1 public class ShareniuLifecycleListener implements ProcessEngineLifecycleListener {
2     public void onProcessEngineBuilt(ProcessEngine processEngine) {
3         System.out.println(processEngine);        //获取 processEngine 对象并输出
4     }
5     public void onProcessEngineClosed(ProcessEngine processEngine) {
6         System.out.println(processEngine);        //获取 processEngine 对象并输出
7     }
8 }
```

定义了 ShareniuLifecycleListener 类之后,接下来将其注入引擎配置类,如代码清单 2-31 所示。

代码清单 2-31 activiti.cfg.xml

```
1 <bean id="processEngineConfiguration"
2     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <property name="dataSource" ref="dataSource"></property>
4     <!-- 自定义生命周期监听类 -->
5     <property name="processEngineLifecycleListener" ref="a"></property>
6 </bean>
7 <bean id="a" class="com.shareniu.chapter2.lifecyclelistener.ShareniuLifecycleListener">
8 </bean>
```

请自行测试以上代码并观察程序的输出。

2.9 其他方式构造引擎实例

上文分析了 ProcessEngines 类中的静态方法 getDefaultProcessEngine 实例化流程引擎的过程,但是这种创建方式存在如下两个缺陷。

(1) activiti.cfg.xml 或者 activiti-context.xml 文件必须位于项目的 classpath 根目录中,否则加载不到。

(2) activiti.cfg.xml 文件中流程引擎配置类的 bean 的 id 值必须是 processEngineConfiguration(默认情况下)。

看到以上两个问题,难免会有疑问:配置文件的路径可以灵活变动吗?配置文件的名称可以修改吗?配置文件中流程引擎配置类的 bean 的 id 值可以是任意值吗?这也是接下来重点讲解的地方。

2.9.1 ProcessEngineConfiguration 类创建引擎

ProcessEngineConfiguration 类中提供了一系列创建流程引擎配置类实例对象的静态

方法,从而方便客户端获取流程引擎实例对象,而无需关心其内部实现细节,这些方法的定义如代码清单 2-32 所示。

代码清单 2-32 ProcessEngineConfiguration.java

```

1 public static ProcessEngineConfiguration createProcessEngineConfigurationFromResourceDefault() {
2     return createProcessEngineConfigurationFromResource("activiti.cfg.xml", "processEngine-
    Configuration");
3 }
4 public static ProcessEngineConfiguration createProcessEngineConfigurationFromResource
    (String resource) {
5     return createProcessEngineConfigurationFromResource(resource, "processEngineConfiguration");
6 }
7 public static ProcessEngineConfiguration createProcessEngineConfigurationFromResource
    (String resource, String beanName) {
8     return BeansConfigurationHelper.parseProcessEngineConfigurationFromResource(resource,
    beanName);
9 }
10 public static ProcessEngineConfiguration createProcessEngineConfigurationFromInputStream
    (InputStream inputStream) {
11     return createProcessEngineConfigurationFromInputStream(inputStream, "processEngine-
    Configuration");
12 }
13 public static ProcessEngineConfiguration createProcessEngineConfigurationFromInputStream
    (InputStream inputStream, String beanName) {
14     return BeansConfigurationHelper.parseProcessEngineConfigurationFromInputStream(inputStream,
    beanName);
15 }
16 public static ProcessEngineConfiguration createStandaloneProcessEngineConfiguration() {
17     return new StandaloneProcessEngineConfiguration();
18 }
19 public static ProcessEngineConfiguration createStandaloneInMemProcessEngineConfiguration() {
20     return new StandaloneInMemProcessEngineConfiguration();
21 }

```

Activiti 在 ProcessEngineConfiguration 类中为开发者提供了丰富多彩的静态方法,用于创建流程引擎配置类实例对象,下面对该类中的方法进行总结说明。

(1) 第 1 行定义的 createProcessEngineConfigurationFromResourceDefault 方法:该方法直接调用第 7 行定义的方法进行处理,并传入第 7 行定义的方法需要的两个输入参数值,通过分析传入的参数值可以得知,该方式构造流程引擎配置类实例需要的配置文件名称必须为 activiti.cfg.xml 并且位于 classpath 根目录中,配置文件中流程引擎配置类的 bean 的 id 值必须是 processEngineConfiguration。

(2) 第 4 行定义的 createProcessEngineConfigurationFromResource 方法:该方法直接调用第 7 行定义的方法进行处理,并传入第 7 行定义的方法需要的两个输入参数值,其中 resource 参数值客户端可以自定义,beanName 为 processEngineConfiguration 对应配置文件中流程引擎配置类的 bean 的 id 值。

(3) 第 7 行定义的 createProcessEngineConfigurationFromResource 方法:该方法比较

灵活, resource 和 beanName 两个输入参数值客户端均可自定义。

(4) 第 10 行定义的 createProcessEngineConfigurationFromInputStream 方法: inputStream 参数为配置文件的数据流, beanName 必须是 processEngineConfiguration(不需要传递)。

(5) 第 13 行定义的 createProcessEngineConfigurationFromInputStream 方法: inputStream 参数为配置文件的数据流, beanName 对应配置文件中流程引擎配置类中 bean 的 id 值, 两者均可自定义。

(6) 第 16 行定义的 createStandaloneProcessEngineConfiguration 方法: 创建 StandaloneProcessEngineConfiguration 实例对象。

(7) 第 19 行定义的 createStandaloneInMemProcessEngineConfiguration 方法: 该方法主要用于创建 StandaloneInMemProcessEngineConfiguration 实例对象。

看到上面一系列的方法, 可能会有这样的疑问: 以上罗列的所有方法均是创建 ProcessEngineConfiguration 实例对象, 并没有提供构造 ProcessEngine 实例对象的方法。其实这样的设计也不难理解, 只要能够获取到 ProcessEngineConfiguration 实例对象, 就可以直接调用该实例对象的 buildProcessEngine 方法创建 ProcessEngine 实例。

2.9.2 编程方式创建引擎

以上几种方式均是通过配置文件创建 ProcessEngineConfiguration 实例对象, 进而通过该实例对象创建 ProcessEngine 实例对象, 接下来讲解如何使用编程方式创建流程引擎实例, 具体实现如代码清单 2-33 所示。

代码清单 2-33 App.java

```
1 public void buildProcessEngine(){
2     ProcessEngineConfiguration processEngineConfiguration = ProcessEngineConfiguration.
createStandaloneProcessEngineConfiguration();
3     //数据库驱动信息
4     processEngineConfiguration.setJdbcDriver("com.mysql.jdbc.Driver");
5     //数据库连接字符串
6     processEngineConfiguration.setJdbcUrl("jdbc:mysql://127.0.0.1:3306/shareniu");
7     //数据库用户名
8     processEngineConfiguration.setJdbcUsername("root");
9     //数据库密码
10    processEngineConfiguration.setJdbcPassword("");
11    ProcessEngine processEngine = processEngineConfiguration.buildProcessEngine();
12 }
```

上面代码的实现逻辑非常简单, 首先第 2 行创建 ProcessEngineConfiguration 实例对象, 然后第 4~10 行为该实例对象填充属性值, 最终第 11 行调用该实例对象的 buildProcessEngine 方法完成引擎实例的创建工作。

第3章

初识流程资源部署

在使用 Activiti 的时候,通常情况下需要根据业务需求绘制流程文档,然后将流程文档进行部署,从而进行后续的一系列操作。部署资源无疑至关重要,如何合理使用引擎提供的部署资源功能,也是前期设计的中中之重,该过程可以检查定义的流程资源是否合理、通用以及校验流程资源格式是否正确等,从而对流程资源进行把关,规避一些不可预知的风险点。

3.1 流程资源概述

本节主要讲解流程资源部署及其相关源码,包含如何部署、部署到何地、部署的流程资源怎么进行删除和更新操作、如何选择最优部署方案等,通常所说的流程资源包含如下。

- (1) 流程文档: 流程文档的扩展名一般为 bpmn20.xml 或者 bpmn, 例如 shareniu.bpmn20.xml。
- (2) 图片: 根据流程文档内容生成的图片, 扩展名一般为 Png。
- (3) Drools 规则文件: 通常情况下扩展名为 drl。
- (4) Form 表单文件: 通常情况下扩展名为 form。

约定

本书中如果没有特殊说明,流程资源等价于流程文档。

3.1.1 流程文档部署生命周期

在开始学习流程文档部署之前,先了解流程文档部署的生命周期,如图 3-1 所示。根据图 3-1,可以把流程文档部署的生命周期分为四大步骤。

- (1) 定义流程文档: 客户端可以根据自己的业务需求定义流程文档。

(2) 启动流程引擎：流程引擎启动之后会自动构造 ProcessEngine 实例对象，这样客户端就可以通过该实例对象获取各种各样的服务类实例对象。例如，TaskService 实例对象，这一系列的服务类实例对象为客户端的操作提供便利。

(3) 部署流程文档：调用流程文档部署命令进行部署，该过程只需要客户端调用部署流程文档的命令即可，流程引擎收到命令之后开始进行如下操作。

- 将流程文档中定义的元素解析为 Activiti 的内部表示 BaseElement 实例。

- 对 BaseElement 实例对象再次解析，进而将其转化为流程虚拟机中的 ActivityImpl 实例对象或者 TransitionImpl 实例对象，该过程非常重要，也是将 BaseElement 实例对象注入流程虚拟机的过程。

(4) 添加缓存：以上所有步骤完成之后，缓存流程定义信息，这样后续节点运转的时候，只需从缓存中取值即可，无须再次执行以上的步骤，从而大幅提升性能，流程引擎默认开启了缓存功能，后续章节会深入讲解缓存机制。

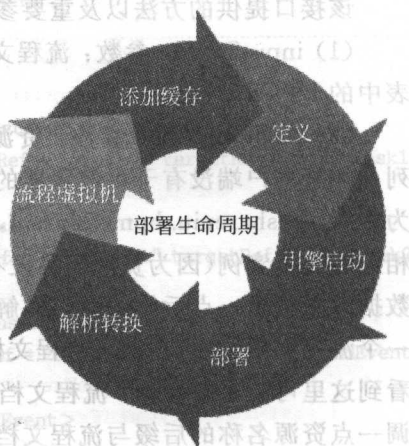


图 3-1 流程文档部署生命周期

3.1.2 DeploymentBuilder 核心类

接下来详细讲解流程文档部署操作，首先明确一点，不管开发人员使用何种方式部署流程文档，最终均是通过流程引擎提供的 API 进行操作。流程文档的部署步骤是首先通过 ProcessEngine 接口的实现类获取到部署服务类 RepositoryService 实例对象，然后借助该实例对象中的 createDeployment 方法创建 DeploymentBuilder 实例对象，DeploymentBuilder 接口的默认实现类为 DeploymentBuilderImpl，经过前面的一系列步骤之后，就可以获取 DeploymentBuilder 类型的实例对象，然后通过该实例对象调用不同的资源部署 API 进行流程资源的部署操作。首先分析 DeploymentBuilder 接口的定义，如代码清单 3-1 所示。

代码清单 3-1 DeploymentBuilder.java

```
1 // 通过 inputStream 流方式部署资源文件
2 DeploymentBuilder addInputStream(String resourceName, InputStream inputStream);
3 // 通过资源文件所在的 classpath 进行部署
4 DeploymentBuilder addClasspathResource(String resource);
5 // 通过字符串的方式部署
6 DeploymentBuilder addString(String resourceName, String text);
7 // 通过 zipInputStream 流部署
8 DeploymentBuilder addZipInputStream(ZipInputStream zipInputStream);
9 // 通过 bpmnModel 对象进行部署
10 DeploymentBuilder addBpmnModel(String resourceName, BpmnModel bpmnModel);
11 // 根据提供的部署方式进行资源的部署
12 Deployment deploy();
```


该接口提供的方法以及重要参数说明如下。

(1) inputStream 参数：流程文档最终的数据流，对应数据库 ACT_GE_BYTEARRAY 表中的 BYTES_列。

(2) resourceName 参数：资源名称，对应 ACT_GE_BYTEARRAY 表中的 NAME_列。如果客户端没有干预部署器的初始化，则资源名称必须以 bpmn20.xml 或者 bpmn 作为后缀，如 shareniu.bpmn20.xml，否则流程文档无法完整部署，进而导致客户端无法启用相应的流程实例（因为资源名称没有被引擎验证通过，ACT_RE_PROCDEF 表中不会产生数据），关于这一点后续的源码讲解会详细阐述该内部处理机制，如果有兴趣可以尝试定义一个流程文档，然后在部署流程文档时显式指定资源名称，进而观察数据库中的数据变化。看到这里可能会有疑惑：流程文档的后缀应该是 bpmn20.xml 或者 bpmn，在这里需要强调一点资源名称的后缀与流程文档的后缀名完全是两个概念，在部署流程文档时，必须确保资源名称的后缀如上所说，需要注意的是如果使用 addZipInputStream 方式部署流程资源，则需要确保流程文档的后缀为 bpmn20.xml 或者 bpmn，因为该方式部署资源时不能显式指定资源名称，所以使用该方式部署资源时，资源名称默认为压缩包中的流程文档名称，相关实现逻辑可以跟进 DeploymentBuilderImpl 类中的 addZipInputStream 方法。

(3) addInputStream 方法：使用数据流的方式部署流程资源（流程文档和流程定义图片等）。

(4) addClasspathResource 方法：使用 classpath 方式直接读取项目中 classpath 目录下指定的流程资源进行部署。

(5) addZipInputStream 方法：将流程资源文件（流程文档和流程定义图片等）打包部署。例如，将流程文档 XML 和生成的流程定义图片，一般是 Png 格式（非必须）打包为 zip 或者 bar 格式。该方式支持多个流程资源文件一次性打包部署（批量部署）。

(6) addString 方式：通过字符串方式部署流程文档。例如，定义一个流程文档之后，将文档的内容读取出来然后直接使用该方式进行部署。

(7) addBpmnModel 方式：通过构造 BpmnModel 实例对象进行流程资源的部署。该方式比较适合开发人员自定义流程设计器，可控性和扩展性比较强。

(8) deploy 方法：流程资源部署的核心方法，负责流程资源的部署。

3.2 流程文档部署

3.2.1 定义流程文档

接下来深入学习流程文档部署的相关操作，需要了解一点，虽然使用几种不同的方式部署流程文档，但操作的流程文档均为同一个，这里对流程文档的内容进行统一描述，流程文档内容如代码清单 3-2 所示，根据该流程文档生成的图片如图 3-2 所示。



图 3-2 流程定义生成的图片

代码清单 3-2 common.bpmn

```
1 <!-- 流程定义中所有的元素 id 值必须全局唯一 -->
2 <process id = "process1" isExecutable = "true">
3   <startEvent id = "start1" name = "开始节点"></startEvent>
4   <sequenceFlow id = "flow1" name = "flow1" sourceRef = "start1" targetRef = "userTask1">
</sequenceFlow>
5   <userTask id = "userTask1" name = "任务节点 1"></userTask>
6   <sequenceFlow id = "flow2" name = "flow2" sourceRef = "userTask1" targetRef = "userTask2">
</sequenceFlow>
7   <userTask id = "userTask2" name = "任务节点 2"></userTask>
8   <sequenceFlow id = "flow3" name = "flow3" sourceRef = "userTask2" targetRef = "endEvent">
</sequenceFlow>
9   <endEvent id = "endEvent" name = "结束节点"></endEvent>
10 </process>
```

该流程文档的定义非常简单，一个开始节点、两个任务节点以及结束节点。接下来重点讲解该流程文档的部署操作。

注意

activiti.cfg.xml 文件的配置，可以参考 2.1.1 节。流程文档中的元素 id 值必须全局唯一。

3.2.2 文本方式部署

addString 方式部署流程文档的常用场景为：流程文档的内容大部分是固定不变的，只有少部分属性在流程文档部署时需要跟外部程序进行交互从而动态填充。例如，开发人员使用图形化工具绘制流程文档，有可能人员组织机构或者其他信息需要从数据库中动态查询，这时就可以使用该方式并结合模板引擎技术动态渲染数据，常用的模板引擎框架有 Velocity、FreeMarker 等，然后生成预期的流程文档内容。该方式就是客户端自定义流程设计器与原生设计器的一种过渡解决方案，具体实现如代码清单 3-3 所示。

代码清单 3-3 DeploymentBuilderTest.java

```
1 public void addString(){
2   String resource = "sharenui.bpmn";
3   //读取文件并获取流程文档中的 xml 信息
4   String text = readTxtFile("E:/learing/src/main/java/com/sharenui/chapter3/common.bpmn");
5   //直接调用 addString 进行资源文件的部署，resource 参数为资源名称
6   DeploymentBuilder deploymentBuilder =
7     repositoryService.createDeployment().addString(resource, text);
8   Deployment deploy = deploymentBuilder.deploy();
9 }
```

该案例使用绝对路径方式获取流程文档的数据流，获取数据流的具体实现如代码 3-4 所示。

代码清单 3-4 DeploymentBuilderTest.java

```

1 public static String readTxtFile(String filePath) {
2     StringBuffer stringBuffer = new StringBuffer();
3     InputStreamReader read = null;
4     try {
5         String encoding = "UTF-8"; //UTF-8 编码
6         File file = new File(filePath);
7         if(file.isFile() && file.exists()){ //判断文件是否存在
8             read = new InputStreamReader(new FileInputStream(file),encoding); //编码格式
9             BufferedReader bufferedReader = new BufferedReader(read);
10            String lineTxt = null;
11            while((lineTxt = bufferedReader.readLine()) != null){
12                stringBuffer.append(lineTxt);
13            }
14            return stringBuffer.toString();
15        }else{
16        }
17        } catch (Exception e) {
18        }finally{
19            try {
20                read.close();
21            } catch (IOException e) {
22            }
23        }
24        return "";
25 }

```

3.2.3 classpath 资源部署

addClasspathResource 方式部署流程文档：该方式会读取项目工程中 classpath 路径下的流程文档。使用该方式部署流程文档会使流程文档与项目产生高耦合，因此不建议在正式环境中使用。需要说明一点，如果使用该方式获取流程文档数据流则需使用/的方式对包名进行分割，形如 com/shareniu/chapter3/common.bpmn，对应的文件结构如图 3-3 所示。

该方式的相关实现如代码清单 3-5 所示。

代码清单 3-5 DeploymentBuilderTest.java

```

1 public void addClasspathResource(){
2     String r = "com/shareniu/chapter3/common.bpmn"; //流程文档的位置
3     DeploymentBuilder d = repositoryService.createDeployment().addClasspathResource(r);
4     Deployment deploy = d.deploy(); //部署
5 }

```

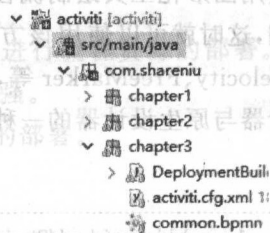


图 3-3 包路径下文件流的获取

3.2.4 流式部署

3.2.4.1 inputStream 部署

接下来使用 addInputStream 方式部署流程文档,具体实现如代码清单 3-6 所示。

代码清单 3-6 DeploymentBuilderTest.java

```
1 public void addInputStreamTest(){
2     //获取流程文档文件流
3     InputStream inputStream = DeploymentBuilderTest.class
4     .getClassLoader().getResource("com/shareniu/chapter3/common.bpmn").openStream();
5     String resource = "shareniu.bpmn";           //资源名称
6     //构造 DeploymentBuilder 对象
7     DeploymentBuilder db = repositoryService.createDeployment().addInputStream(resourceName,
8     in);
9     //部署操作
10    Deployment deploy = db.deploy();
11 }
```

在上述代码中,第 3~4 行直接使用类加载器获取 common.bpmn 文件的数据流,获取到数据流之后直接构造 DeploymentBuilder 实例对象,然后调用该实例对象的 deploy() 方法部署流程文档。

3.2.4.2 zipInputStream 部署

以上三种部署方式,一次只能部署一个流程文档,如果期望一次性部署多个流程文档,很显然上面的部署方式是不支持的,但是很幸运,Activiti 提供了打包部署机制,可以把多个流程文档以及流程文档对应的图片或者表单等统一打包为.zip 的压缩文件(一般使用这种方式)或者 .bar 压缩文件,然后再对其进行部署,打包之后的结构如图 3-4 所示,相关实现如代码清单 3-7 所示。

代码清单 3-7 DeploymentBuilderTest.java

```
1 InputStream inputStream = ProcessEnginesDemo.class.getClassLoader().getResourceAsStream("demo1.
zip");
2 ZipInputStream zipInputStream = new ZipInputStream(inputStream);
3 Deployment deploy = repositoryService2.createDeployment().addZipInputStream(zipInputStream).
deploy();
```

使用 addZipInputStream 方式部署流程资源,需要将打包的流程文档或者图片等存放在压缩包中,其内部使用迭代器方式循环遍历压缩包中的文件并读取相应的文件流。

建议

可以跟进 DeploymentBuilderImpl 类中的 addZipInputStream 方法查看其处理机制。

3.3 BpmnModel 方式部署

使用该方式复杂点在于客户端需要手动构造流程引擎中的 BpmnModel 实例对象,如果开发人员平时设计流程文档时过多依赖图形化工具,可能对流程文档中定义的元素含义与引擎内部相对应的元素属性承载类不熟悉,因此使用该方式可能有点棘手,显得力不从心,本节先写一个简单的入门程序,该案例的相关实现如代码清单 3-8 所示。

代码清单 3-8 DeploymentBuilderTest.java

```

1 public void addBpmnModel(){
2     SequenceFlow flow1 = new SequenceFlow();
3     flow1.setId("flow1");
4     flow1.setName("开始节点->任务节点 1");
5     flow1.setSourceRef("start1");
6     flow1.setTargetRef("userTask1");
7     //flow2 的名称为"任务节点 1->任务节点 2"
8     SequenceFlow flow2 = new SequenceFlow();
9     flow2.setId("flow2");
10    flow2.setName("任务节点 1->任务节点 2");
11    flow2.setSourceRef("userTask1");
12    flow2.setTargetRef("userTask2");
13    //flow3 的名称为"任务节点 2->endEvent"
14    SequenceFlow flow3 = new SequenceFlow();
15    flow3.setId("flow3");
16    flow3.setName("任务节点 2->结束节点");
17    flow3.setSourceRef("userTask2");
18    flow3.setTargetRef("endEvent");
19    String resource = "shareniu_addBpmnModel";
20    //实例化 BpmnModel 类
21    BpmnModel bpmnModel = new BpmnModel();
22    //实例化 Process 类 一个 BpmnModel 实例对象可以包含多个 Process 实例对象
23    Process process = new Process();
24    process.setId("process1");
25    //封装开始节点
26    StartEvent start = new StartEvent();
27    start.setName("开始节点");
28    start.setId("start1");
29    start.setOutgoingFlows(Arrays.asList(flow1));
30    //任务节点 1
31    UserTask userTask1 = new UserTask();
32    userTask1.setName("任务节点 1");
33    userTask1.setId("userTask1");
34    userTask1.setIncomingFlows(Arrays.asList(flow1));
35    userTask1.setOutgoingFlows(Arrays.asList(flow2));
36    //任务节点 2

```

```

37 UserTask userTask2 = new UserTask();
38 userTask2.setName("任务节点 2");
39 userTask2.setId("userTask2");
40 userTask2.setIncomingFlows(Arrays.asList(flow2));
41 userTask2.setOutgoingFlows(Arrays.asList(flow3));
42 //结束节点
43 EndEvent endEvent = new EndEvent();
44 endEvent.setName("结束节点");
45 endEvent.setId("endEvent");
46 endEvent.setIncomingFlows(Arrays.asList(flow3));
47 //将所有的 FlowElement 添加到 process 中
48 process.addFlowElement(start);
49 process.addFlowElement(flow1);
50 process.addFlowElement(userTask1);
51 process.addFlowElement(flow2);
52 process.addFlowElement(userTask2);
53 process.addFlowElement(flow3);
54 process.addFlowElement(endEvent);
55 bpmnModel.addProcess(process);
56 DeploymentBuilder deploymentBuilder =
57 repositoryService.createDeployment().addBpmnModel(resource, bpmnModel);
58 Deployment deploy = deploymentBuilder.deploy();
59 }

```

在上述代码中,第2~6行构造 flow1 对象,并为其填充属性。第8~12行构造 flow2 对象,并为其填充属性。第14~18行构造 flow3 对象,并为其填充属性。第23行实例化 Process 类。第37~41行构造 userTask2 对象,并为其填充属性;第43~46行构造 endEvent 对象,并为其填充属性。第48~54行将上述创建的一系列对象设置到 process 对象中。下面使用图 3-5 对上述代码中涉及的流程进行通俗易懂的描述。

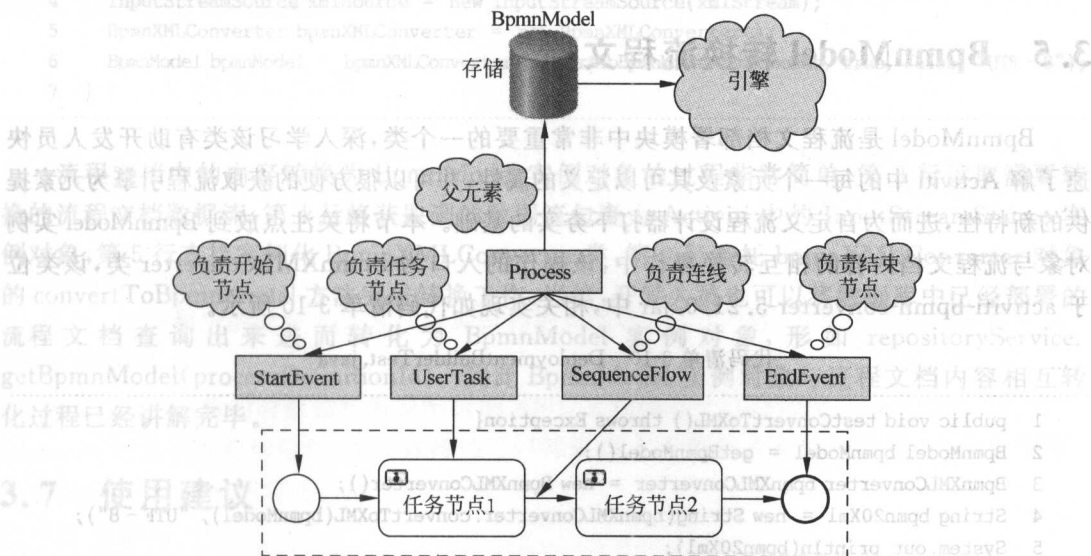


图 3-5 BpmnModel 描述

3.4 校验 BpmnModel 实例对象

代码 3-8 中手工构建了 BpmnModel 实例对象并对其进行部署,但也同时面临一个问题,如何才能确保该实例对象转换之后的 XML 格式是正确的呢?接下来详细讲解如何校验 BpmnModel 实例对象,校验该实例对象需要使用 ProcessValidatorFactory 类,该类位于 activiti-process-validation-5.21.0.jar 中,具体实现如代码清单 3-9 所示。

代码清单 3-9 DeploymentBuilderTest.java

```
1 public void testProcessValidator(){
2     BpmnModel bpmnModel = getBpmnModel();
3     ProcessValidatorFactory processValidatorFactory = new ProcessValidatorFactory();
4     ProcessValidator defaultProcessValidator =
5     processValidatorFactory.createDefaultProcessValidator();
6     List<ValidationError> validate = defaultProcessValidator.validate(bpmnModel);
7     System.out.println(validate.size());
8 }
```

在上述代码中,第 2 行获取 BpmnModel 实例对象,该实例对象的构造过程可以参考 3.3 节。第 3 行直接实例化 ProcessValidatorFactory 类,第 4~5 行通过该工厂类创建流程验证器对象 defaultProcessValidator,第 6 行委托 defaultProcessValidator 对象中的 validate 方法校验 bpmnModel 对象,该方法的返回值为 ValidationError 集合(封装验证之后的信息),如果该集合长度为 0 则说明 bpmnmodel 对象已经成功通过校验,否则没有验证通过。

流程文档的格式验证是必不可少的一个重要环节,如果流程文档部署之前开启了格式校验,则可以避免将错误或者不可使用的流程文档部署到引擎表中,从而规避一系列可能出现的问题,防患于未然。

3.5 BpmnModel 转换流程文档

BpmnModel 是流程文档部署模块中非常重要的一个类,深入学习该类有助开发人员快速了解 Activiti 中的每一个元素及其可以定义的属性,并可以很方便的获取流程引擎为元素提供的新特性,进而为自定义流程设计器打下夯实的基础。本节将关注点放到 BpmnModel 实例对象与流程文档内容的相互转换操作中,该操作的入口为 BpmnXMLConverter 类,该类位于 activiti-bpmn-converter-5.21.0.jar 中,相关实现如代码清单 3-10 所示。

代码清单 3-10 DeploymentBuilderTest.java

```
1 public void testConvertToXML() throws Exception{
2     BpmnModel bpmnModel = getBpmnModel();
3     BpmnXMLConverter bpmnXMLConverter = new BpmnXMLConverter();
4     String bpmn20Xml = new String(bpmnXMLConverter.convertToXML(bpmnModel), "UTF-8");
5     System.out.println(bpmn20Xml);
6 }
```

在上述代码中,第3行实例化 BpmnXMLConverter 类,第4行调用 bpmnXMLConverter 对象的 convertToXML 方法将 BpmnModel 实例对象转换为 XML 内容,第5行输出转换之后的 XML 内容。以上步骤执行完后,可以看到输出结果如代码清单 3-11 所示。

代码清单 3-11 BpmnModel 转化为 XML 内容

```
1 <?xml version = "1.0" encoding = "UTF - 8"?>
2 <process id = "process1" isExecutable = "true">
3 <startEvent id = "start1" name = "开始节点"></startEvent>
4 <sequenceFlow id = "flow1" name = "开始节点任务节点 1" sourceRef = "start1"
5 targetRef = "userTask1"></sequenceFlow>
6 <userTask id = "userTask1" name = "任务节点 1"></userTask>
7 <sequenceFlow id = "flow2" name = "任务节点 2" sourceRef = "userTask1"
8 targetRef = "userTask2"></sequenceFlow>
9 <userTask id = "userTask2" name = "任务节点 2"></userTask>
10 <sequenceFlow id = "flow3" name = "结束节点" sourceRef = "userTask2"
11 targetRef = "endEvent"></sequenceFlow>
12 <endEvent id = "endEvent" name = "结束节点"></endEvent>
13 </process>
```

3.6 流程文档转换 BpmnModel

接下来讲解如何把流程文档中的内容转化为 BpmnModel 实例对象,相关实现如代码清单 3-12 所示。

代码清单 3-12 DeploymentBuilderTest.java

```
1 public void testConvertToBpmnModel() throws Exception{
2     String resource = "com/shareniu/chapter3/common.bpmn";
3     InputStream xmlStream = this.getClass().getClassLoader().getResourceAsStream(resource);
4     InputStreamSource xmlSource = new InputStreamSource(xmlStream);
5     BpmnXMLConverter bpmnXMLConverter = new BpmnXMLConverter();
6     BpmnModel bpmnModel = bpmnXMLConverter.convertToBpmnModel(xmlSource, true, false, "UTF - 8");
7 }
```

流程文档中的内容转换为 BpmnModel 实例对象的过程非常简单,第3行读取需要转换的流程文档数据流,第4行将获取到的数据流包裹为 Activiti 中的 InputStreamSource 实例对象,第5行直接实例化 BpmnXMLConverter 类,第6行委托 bpmnXMLConverter 对象的 convertToBpmnModel 方法完成转换工作,当然,开发人员也可以将数据库中已经部署的流程文档查询出来进而转化为 BpmnModel 实例对象,形如 repositoryService.getBpmnModel(processDefinitionId)。到此 BpmnModel 实例对象与流程文档内容相互转化过程已经讲解完毕。

3.7 使用建议

对上述不同方式部署流程文档的使用场景再次总结如下。

(1) 如果客户端需要开发一套流程设计器建议使用 addBpmnModel 方式部署,该方式

(2) 如果项目中流程文档的大部分内容是固定的,只有极少数属性需要在流程部署时动态从数据库中读取,建议使用 `addString` 方式进行部署,并可以配合 `Velocity` 等静态化框架一起使用。

(4) `addClasspathResource` 方式不建议在正式环境中使用,使用该方式会使流程文档与项目高耦合,且不灵活。

以上几种部署方式可以逐个尝试使用,并根据不同的使用场景灵活运用。

第4章

流程文档解析原理

在使用 Activiti 的时候,会经常根据不同的业务场景绘制不同的流程文档,绘制流程文档的过程中需要使用大量元素,例如任务节点(userTask)、连线(sequenceFlow)等,那么 Activiti 是如何解析元素的呢?如果 Activiti 中提供的元素不能完全满足业务需求,又该如何为元素扩展属性?如何让扩展的元素拥有更高的通用性?这就涉及了 Activiti 中的元素解析,正所谓了解其本质,看透其实现原理才能更好地对 Activiti 解析元素的功能进行扩展,所以本章重点讲解 Activiti 解析元素的整个过程以及框架预留给开发人员的扩展点,从而使开发人员可以结合自身业务对元素属性进行灵活扩展和改造。

4.1 文档解析基础

4.1.1 文档解析模型

关于 XML 文档解析技术,目前经常使用的有 DOM、SAX、DOM4J 等,这些解析方式从模型上可以划分为如下两种:DOM(文档对象)模型和流模型,下面对比这两种模型的优缺点。

(1) DOM 模型。

优点:文档解析的时候允许客户端编辑和更新 XML 文档内容,并可随机访问文档中定义的元素数据。

缺点:文档解析的时候会将需要解析的 XML 文档内容一次性加载到内存,进而映射为 Document 对象中的树形结构,基于该特性可以知道该方式解析大文件的时候,内存占用率大,元素遍历查找慢,很容易造成性能问题。

(2) 流模型。

优点:该方式解析文档的时候,每一次操作只会将需要解析的节点放置到内存中,从头部开始,读取一段,处理一段,这样就解决了文档对象模型可能引发的性能问题,该方式内存

占用率少,性能大幅提升。

缺点: 该方式解析文档的时候文档是只读的,不可以编辑,文件流只能前进不能后退。

4.1.2 Activiti 文档解析技术选型演变

基于上面所说的两种解析模型,因为 Activiti 将流程文档完全交给客户端定义和使用,所以最终生成的 XML 文件大小完全由用户决定,是不可控因素之一,软件架构设计之初应该尽量屏蔽这种不可控的风险,所以 Activiti 最终选择流模型方式进行文档解析工作。

流模型常用的技术有 SAX 和 STAX 两种,SAX 使用的是推模型,STAX 使用的是拉模型。

(1) 推模型。

推模型: SAX 处理方式,是一种事件驱动机制,文档解析的时候,每当发现一个元素节点就会触发相应的事件,因此客户端需要编写监听触发事件的处理程序,使用该方式解析文档无疑增加了客户端操作的复杂度,使用起来比较麻烦,不灵活。

(2) 拉模型。

拉模型: STAX 处理方式,文档解析时客户端可以定制自己感兴趣的节点主动从读取器(Reader)中进行拉取,选择性的处理节点事件。对比推模型,该方式灵活性大大提高。

基于上面罗列的流模型解决方案,Activiti 最终选用 STAX 方式解析流程文档。STAX 提供了两套处理 XML 的 API,它们分别是基于指针的 API 和基于迭代器的 API。两者解析原理类似,下面讲解的文档解析实战主要是基于指针的 API,如果对基于迭代器的 API 操作感兴趣可以自行学习。

注意

Activiti 在 5.12.1 版本中开始使用 STAX 方式解析流程文档,之前版本使用的是 SAX 方式解析流程文档。

4.1.3 文档解析实战

为了便于理解,可以把 STAX 模型理解为一个不可回流的流水管道,该管道主要用来存储流程文档定义信息,读取器可以理解为水龙头,每次读取器读取文档内容的时候,客户端可以对读取的流进行处理,也可以不理睬。下面创建一个名为 stax.xml 的文档用于测试,该文档的内容如代码清单 4-1 所示。

代码清单 4-1 stax.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
3   xmlns:activiti="http://activiti.org/bpmn" targetNamespace="shareniu">
4   <process>
5     <!-- userTask 任务节点 -->
6     <userTask id="operationTask" name="operationTask" activiti:assignee="admin">
7       <!-- endEvent 结束节点 -->
```

```

8      <endEvent id="end" name="End"></endEvent>
9  </process>
10 </definitions>

```

在上述代码中,定义了父元素 process 和两个子元素 userTask、endEvent。流程文档定义完毕,再写一个解析类对 stax.xml 内容进行解析,该类的相关定义如代码清单 4-2 所示。

代码清单 4-2 StaxTest.java

```

1  public class StaxTest {
2      public static void main( String[] args ) throws Exception
3      { /* 使用 XMLInputFactory 工厂 */
4          XMLInputFactory xif = XMLInputFactory.newInstance();
5          /* 使用类加载器加载资源文件 */
6          InputStream in =
7              StaxTest.class.getClassLoader().getResourceAsStream( "com/shareniu/chapter4/stax.xml" );
8          /* 用 Reader 读取资源文件流 */
9          Reader reader = new XmlStreamReader( in );
10         /* 根据 reader 创建 XMLStreamReader 实例对象 */
11         XMLStreamReader xtr = xif.createXMLStreamReader( reader );
12         while ( xtr.hasNext() ){
13             int event = xtr.next(); /* 文档开始 */
14             if ( event == XMLStreamConstants.START_DOCUMENT ) {
15             } else if ( event == XMLStreamConstants.END_DOCUMENT ) /* 文档结束 */
16             {
17             } else if ( event == XMLStreamConstants.START_ELEMENT ) /* 开始节点解析 */
18             { /* 获取元素名称 */
19                 System.out.println( "节点开始解析:" + xtr.getLocalName() );
20                 if ( xtr.getLocalName().equals( "userTask" ) || xtr.getLocalName().equals
21                     ( "endEvent" ) ) {
22                     for ( int i = 0; i < xtr.getAttributeCount(); i++ ) {
23                         if ( xtr.getAttributeName( i ).toString().startsWith( "{" ) ) {
24                             /* 第一个参数表示属性所使用的命名空间 */
25                             System.out.print( " --->>>" + xtr.getAttributeValue( "http://
26                                 activiti.org/bpmn", "assignee" ) );
27                         } else {
28                             System.out.print( xtr.getAttributeName( i ) + ":" );
29                             System.out.print( xtr.getAttributeValue( i ) + ";" );
30                         }
31                     } /* 节点结束 */
32                 } else if ( event == XMLStreamConstants.END_ELEMENT )
33                 {
34                 } else if ( event == XMLStreamConstants.CHARACTERS ) {
35                     String text = xtr.getText();
36                     if ( !text.isEmpty() && text.trim().length() > 0 ) {
37                     }
38                 }
39             }
40         }
41     }
42 }

```


下面对上述代码中 STAX 技术解析流程文档的处理逻辑加以总结：第 4 行使用 XMLInputFactory 类的 newInstance 方法创建 XMLInputFactory 实例对象，第 6~7 行获取 XML 文档中的数据流，第 9 行根据获取到的数据流构造读取器 Reader 实例对象，第 11 行调用 xif 对象的 createXMLStreamReader 方法，并传入该方法需要的读取器构造 XMLStreamReader 实例对象，第 13 行中的 event 变量值表示当前指针所指向的标记或者事件的类型，开发人员可以根据标记或者事件类型定制感兴趣的部分进行解析处理。Event 可取值均定义在 XMLStreamConstants 类中，如表 4-1 所示。

表 4-1 事件类型的名称、含义和值

名 称	含 义	值
START_ELEMENT	元素的开始	1
END_ELEMENT	元素的结尾	2
PROCESSING_INSTRUCTION	处理指令	3
CHARACTERS	字符(文本或空格)	4
COMMENT	注释	5
SPACE	可忽略的空格	6
START_DOCUMENT	文档的开始	7
END_DOCUMENT	文档的结尾	8
ENTITY_REFERENCE	实体的引用	9
ATTRIBUTE	元素的属性	10
DTD	DTD	11
CDATA	CDATA 块	12
NAMESPACE	命名空间的声明值为 13	13
NOTATION_DECLARATION	标记的声明值为 14	14
ENTITY_DECLARATION	实体的声明值为 15	15

STAX 方式解析 XML 文档的核心点在于程序可以根据当前的标记或事件类型做出相应的处理。深入理解上面罗列的每一个 event 值的具体含义后，就会发现使用 STAX 技术解析 XML 文档是非常简单的一件事情，因为它的用法和 Java 迭代器(Iterator)是一样的，至此 STAX 技术解析 XML 文档的过程已经分析讲解完毕。

4.2 元素解析功能架构设计

4.2.1 BPMN2.0 元素概述

绘制流程文档时，定义的元素需要遵循 BPMN2.0 规范。BPMN2.0 规范将所有的流程定义元素抽象提取为三大要素如图 4-1 所示。

(1) Event(事件)：流程的创建、流转、结束等均需要事件支持，例如在流程文档绘制阶段，定义开始节点和结束节点是一个必不可少的环节。可通过事件机制为 workflow 系统增加辅助功能。

(2) Gateways(网关)：所谓“网关”就是用来辅助决定流程实例最终流转的目的地，可以用来并行执行节点、也可以作为聚合或者条件分支使用，常用的网关类型有排他网关、并行网关、兼容网关三种。

(3) Activities(活动)：有生命周期的元素或者节点都可以称之为“活动”，例如任务节

点、子流程、引用流程等。活动节点可以作为任何连线元素的源头或者目标。

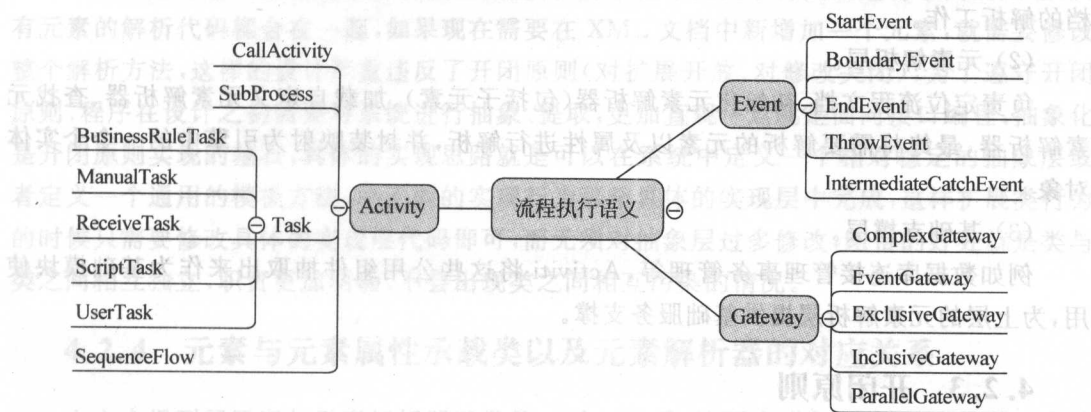


图 4-1 BPMN2.0 三大要素

举个通俗易懂的例子,可以把流程文档的定义过程想象为一个水流管道,水流的发源地、目的地可以理解成事件,水流的分支或者聚合可以理解成网关,水流途径的节点统称为活动,水流途径的管道则可以理解为连线,网关和连线最终决定水流的宏观走向以及运动轨迹。

4.2.2 元素解析功能架构设计

在深入学习流程文档解析之前,首先分析 Activiti 元素解析的功能架构设计图,如图 4-2 所示。

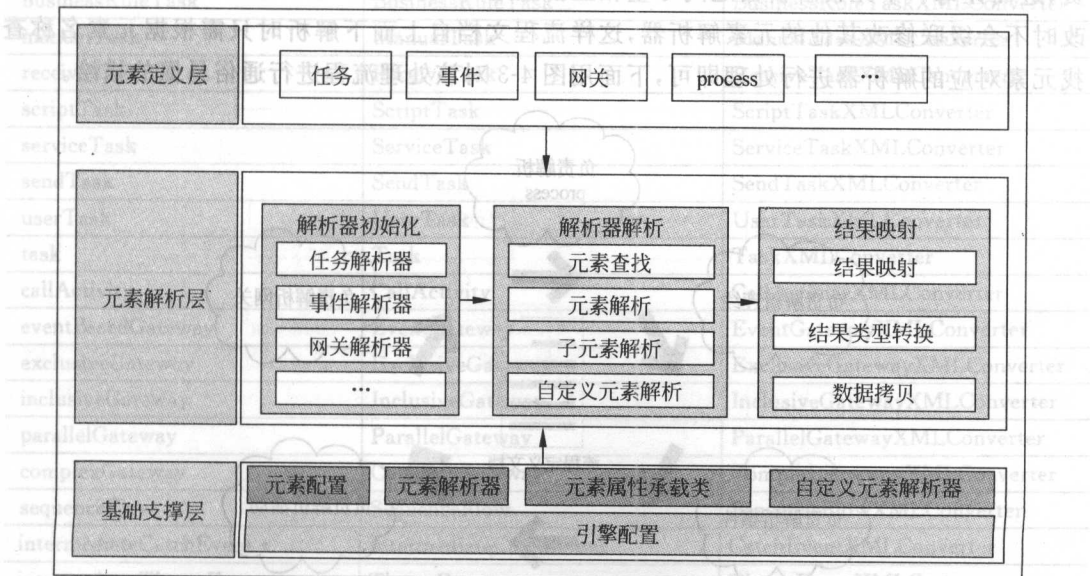


图 4-2 元素解析功能架构设计

根据上图可以把 Activiti 元素解析功能架构分为三层。

(1) 元素定义层。

元素定义层完全交给客户端,开发人员可以结合自己的业务场景组装一系列元素,最终

完成流程文档的定义工作,流程文档定义完毕之后就可以直接调用元素解析层实现流程文档的解析工作。

(2) 元素解析层。

负责定位流程文档、初始化元素解析器(包括子元素)、加载自定义元素解析器、查找元素解析器,最终将需要解析的元素以及属性进行解析,并封装映射为引擎中的一个实体对象。

(3) 基础支撑层。

例如数据库连接管理事务管理等,Activiti 将这些公用组件抽取出来作为基础模块使用,为上层的元素解析层提供基础服务支撑。

4.2.3 开闭原则

如何实现流程文档的解析工作,首先每个元素都需要定义公共属性,例如 id、name 等,可以把类似 id、name 等公共属性的解析封装为一个通用的解析方法,这样程序解析流程文档元素时,只需要调用该方法即可完成公共属性的解析工作。流程文档中还有些元素的属性信息是容易变化的,为什么容易变化呢,因为 Activiti 版本升级时可能会对部分元素添加新特性,此时就需要对扩展元素对应的属性承载类添加一个新的属性定义,更致命的问题是添加新的属性就需要修改该元素的解析代码,为了应对类似这样的变化,Activiti 为每一个元素包括子元素定义了一个对应的元素解析器,从而使每一个元素解析器只负责一个元素的解析工作,这样设计的好处就是使元素与元素解析器一一对应,分离抽取元素解析器的职责,进而让每一个元素解析器之间尽量相互独立运行,当任意一个元素解析器的逻辑需要修改时不会级联修改其他的元素解析器,这样流程文档自上而下解析时只需根据元素名称查找元素对应的解析器进行处理即可,下面用图 4-3 对该处理流程进行通俗易懂的描绘。

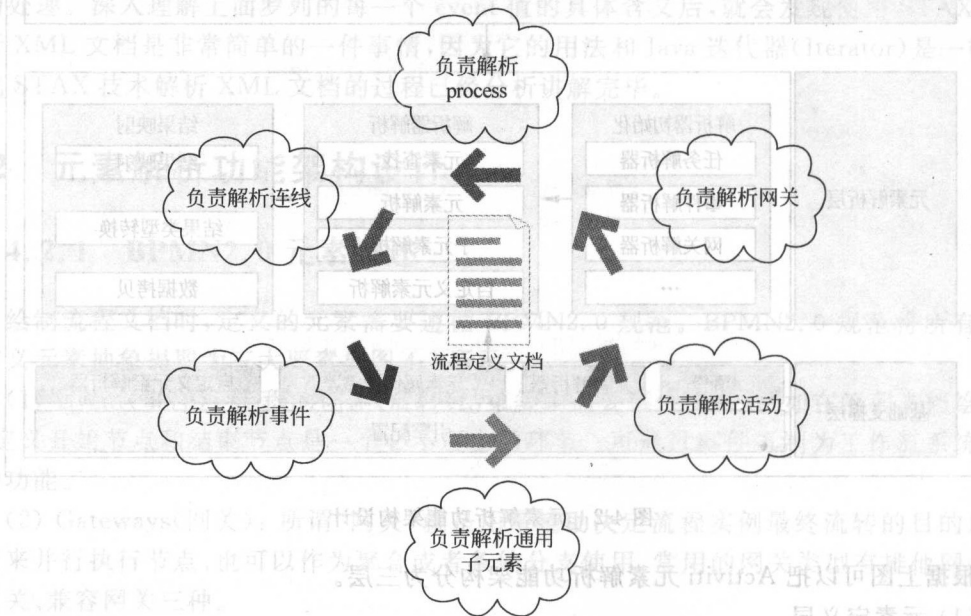


图 4-3 流程文档解析过程通俗的理解

根据图 4-3,可以发现上文写的解析流程文档的代码有一个很大的缺陷,该处理过程中程序做了大量判断当前标记或事件类型的逻辑,并根据不同的事件类型进行不同的处理,所有元素的解析代码糅合在一起,如果现在需要在 XML 文档中新增一个元素,就需要修改整个解析方法,这样的设计严重违反了开闭原则(对扩展开放、对修改关闭),为了遵守开闭原则,程序在设计之初需要对系统进行抽象、提取,更加直观一点就是面向接口编程,抽象化是开闭原则实现的基石,具体的实现思路就是可以在系统中定义一个相对稳定的抽象层或者定义一个通用的模板方法,将不同的实现行为移至具体的实现层中完成,这样扩展类行为的时候只需要修改具体的实现层代码即可,而无须对抽象层过多修改,附带的好处就是类与类之间相互独立,职责更加明确,不会出现类之间相互污染的情况。

4.2.4 元素与元素属性承载类以及元素解析器的对应关系

上文中提到了元素与元素解析器通常是一对一关系,接下来讲解元素以及元素对应的解析器、元素属性承载类(元素解析的同时,需要将已经解析完毕的属性值封装到指定的类中,为了便于区分,将这一系列的类统称为元素属性承载类)三者之间的关系,如表 4-2 所示,如果期望查找某个元素的解析逻辑,可以直接通过表 4-2 查找元素对应的解析器,然后查看元素解析器中的处理逻辑即可。

表 4-2 元素、元素属性承载类和元素解析器对应关系

元素名称	元素属性承载类	元素解析器
endEvent	EndEvent	EndEventXMLConverter
startEvent	StartEvent	StartEventXMLConverter
businessRuleTask	BusinessRuleTask	BusinessRuleTaskXMLConverter
manualTask	ManualTask	ManualTaskXMLConverter
receiveTask	ReceiveTask	ReceiveTaskXMLConverter
scriptTask	ScriptTask	ScriptTaskXMLConverter
serviceTask	ServiceTask	ServiceTaskXMLConverter
sendTask	SendTask	SendTaskXMLConverter
userTask	UserTask	UserTaskXMLConverter
task	Task	TaskXMLConverter
callActivity	CallActivity	CallActivityXMLConverter
eventBasedGateway	EventGateway	EventGatewayXMLConverter
exclusiveGateway	ExclusiveGateway	ExclusiveGatewayXMLConverter
inclusiveGateway	InclusiveGateway	InclusiveGatewayXMLConverter
parallelGateway	ParallelGateway	ParallelGatewayXMLConverter
complexGateway	ComplexGateway	ComplexGatewayXMLConverter
sequenceFlow	SequenceFlow	SequenceFlowXMLConverter
intermediateCatchEvent	IntermediateCatchEvent	CatchEventXMLConverter
intermediateThrowEvent	ThrowEvent	ThrowEventXMLConverter
boundaryEvent	BoundaryEvent	BoundaryEventXMLConverter
textAnnotation	TextAnnotation	TextAnnotationXMLConverter
association	Association	AssociationXMLConverter
dataStoreReference	DataStoreReference	DataStoreReferenceXMLConverter

续表

元素名称	元素属性承载类	元素解析器
dataObject	ValuedDataObject	ValuedDataObjectXMLConverter
subProcess	SubProcess	SubProcessParser
signal	Signal	SignalParser
resource	Resource	ResourceParser
process	Process	ProcessParser
participant	Pool	ParticipantParser
multiInstanceLoopCharacteristics	MultiInstanceLoopCharacteristics	MultiInstanceParser

注意

BaseBpmnXMLConverter 类作为所有元素解析器的基类存在。

4.2.5 元素属性承载类架构

下面重点讲解表 4-2 中的元素属性承载类,首先分析 BaseElement 类的设计架构,如图 4-4 所示。

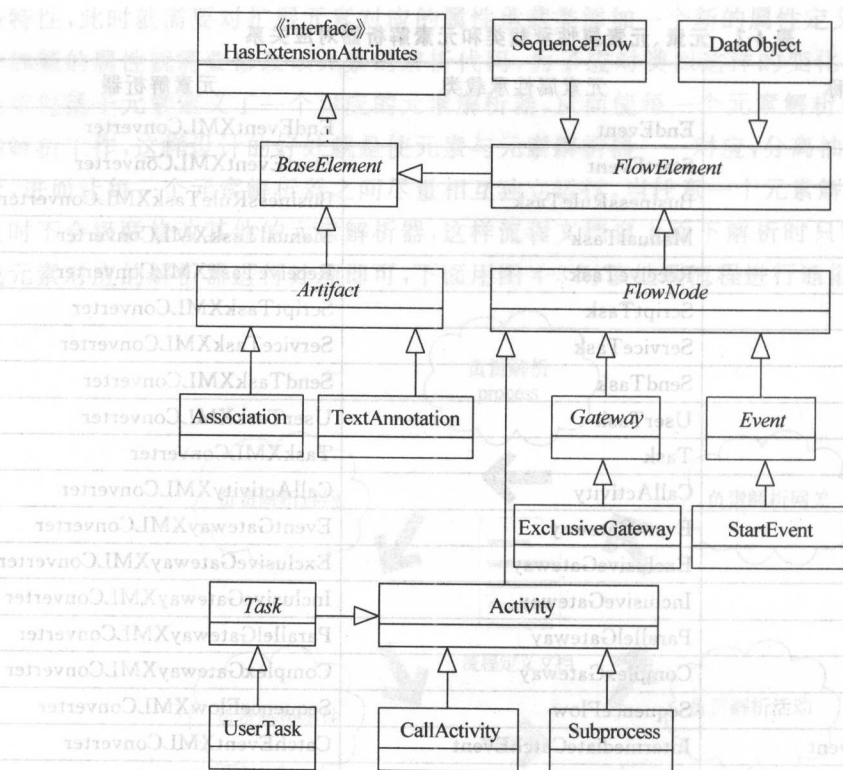


图 4-4 实体对象架构

通过上面的类图,可以很清晰地从全局角度了解 BaseElement 的脉络,后续章节会逐步讲解上面类图中涉及的类,接下来先大致讲解各个类的职责。

- HasExtensionAttributes: 该接口定义了用户自定义属性的存储和获取方法。因为所有的元素属性承载类都实现了该接口,所以可以大胆推测所有的元素都是可以扩展的。
- BaseElement: 该抽象类实现了 HasExtensionAttributes 接口,封装了 id、xmlLineNumber、xmlColumnNumber(元素在 XML 文件中的坐标信息)等属性以及克隆 BaseElement 实例对象的抽象方法。
- FlowElement: 该抽象类继承 BaseElement 类,封装了元素定义的名称(name)、描述信息(documentation)、执行监听器(executionListeners)属性信息以及克隆 FlowElement 实例对象的抽象方法。
- FlowNode: 该抽象类继承 FlowElement 类,并对元素的如下属性进行了定义,异步执行(asynchronous)、独占模式(notExclusive)、节点的出线(incomingFlows)、节点的入线(outgoingFlows)等信息。
- Activity: 该抽象类继承 FlowNode 类,并对元素的如下属性进行了定义,默认连线(defaultFlow)、多实例(loopCharacteristics)等信息,该类为子流程、引用流程以及 Task 节点的基类。
- Task: 该抽象类继承 Activity 类,该类是所有任务类的基类,例如 ScriptTask、UserTask 等。
- SubProcess: 子流程元素属性承载类,子流程中所有的元素都在该类中聚合。
- Event: 该抽象类继承 FlowNode 类,是 StartEvent、EndEvent 等事件类型的基类。
- Gateway: ParallelGateway、InclusiveGateway 等网关的父类。
- SequenceFlow: 对连线信息进行封装,例如条件表达式 conditionExpression 等。
- Artifact: 该抽象类继承 BaseElement 类,该类中定义了克隆 Artifact 实例对象的抽象方法。

4.3 元素解析环境准备

4.3.1 文档转换器

3.6 节详细讲解了流程文档内容与 BpmnModel 实例对象的相互转化,该转换过程涉及流程文档中的元素解析,下面详细讲解引擎解析流程文档的具体处理步骤,首先定义一个流程文档 oneTaskProcess.bpmn20.xml,该流程文档的内容如代码清单 4-3 所示,流程定义图如图 4-5 所示。

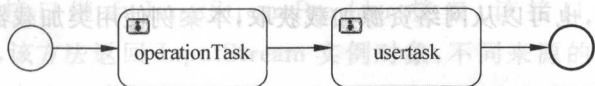


图 4-5 文档定义图

代码清单 4-3 oneTaskProcess.bpmn20.xml

```
1 <!-- 流程的 id 和名称信息 -->
2 <process id="extensionOperationProcess" name="extension" isExecutable="true">
```

```

3 <!-- 开始节点 -->
4 <startEvent id="start" name="Start">
5   <documentation> start </documentation> <!-- 开始节点的文档描述 -->
6 </startEvent>
7 <!-- start 到 operationTask 连线信息 -->
8 <sequenceFlow id="flow1" sourceRef="start" targetRef="operationTask"></sequenceFlow>
9 <userTask id="operationTask" name="operationTask" activiti:assignee="usertask1">
10 </userTask>
11 <!-- operationTask 到 usertask 任务节点的连线信息 -->
12 <sequenceFlow id="flow2" sourceRef="operationTask" targetRef="usertask"></sequenceFlow>
13 <!-- usertask 任务节点定义信息 -->
14 <userTask id="usertask" name="usertask" activiti:assignee="sharenui"></userTask>
15 <!-- usertask 到 end 结束节点连线信息 定义 -->
16 <sequenceFlow id="flow3" sourceRef="usertask" targetRef="end"></sequenceFlow>
17 <!-- end 结束节点的定义 -->
18 <endEvent id="end" name="End"></endEvent>
19</process>

```

上述流程文档中定义了开始节点 start、任务节点 operationTask、usertask 和结束节点 End。

注意

所有元素的属性 id 值必须在流程文档中全局唯一。

下面定义一个类,该类主要用于解析流程文档,如代码清单 4-4 所示。

代码清单 4-4 ExtensionOperationProcessTest.java

```

1 public void convertToBpmnModel() {
2   // 获取流程文档数据流
3   InputStream xmlStream = this.getClass().getClassLoader().getResourceAsStream(
4     "com/sharenui/chapter4/oneTaskProcess.bpmn20.xml");
5   StreamSource xmlSource = new InputSource(xmlStream);
6   BpmnXMLConverter bpmnXMLConverter = new BpmnXMLConverter();
7   BpmnModel bpmnModel = bpmnXMLConverter.convertToBpmnModel(xmlSource,
8     false, false, "UTF-8");
9 }

```

将上面代码中 convertToBpmnModel 方法的处理逻辑进行如下总结。

(1) 第 3~4 行获取流程文档的文件流。流程文档的文件流获取方式有很多,可以使用绝对定位的方式获取,也可以从网络资源加载获取,本案例使用类加载器的方式获取流程文档的文件流。

(2) 包裹文件流。第 5 行将获取到的 InputStream 类型的输入流,然后将其包裹为 Activiti 引擎可识别的 StreamSource。

(3) 第 6 行实例化 BpmnXMLConverter 类,该类非常重要,负责调度元素的解析工作,并维护元素解析器与元素之间的对应关系。

(4) 第 7~8 行调用 BpmnXMLConverter 实例对象的 convertToBpmnModel 方法解析

元素,该方法最终返回 BpmnModel 类型的实例对象 bpmnModel,并将元素解析之后的结果填充到 bpmnModel 对象中。配合下面的时序图(如图 4-6 所示),可能会更加容易理解。

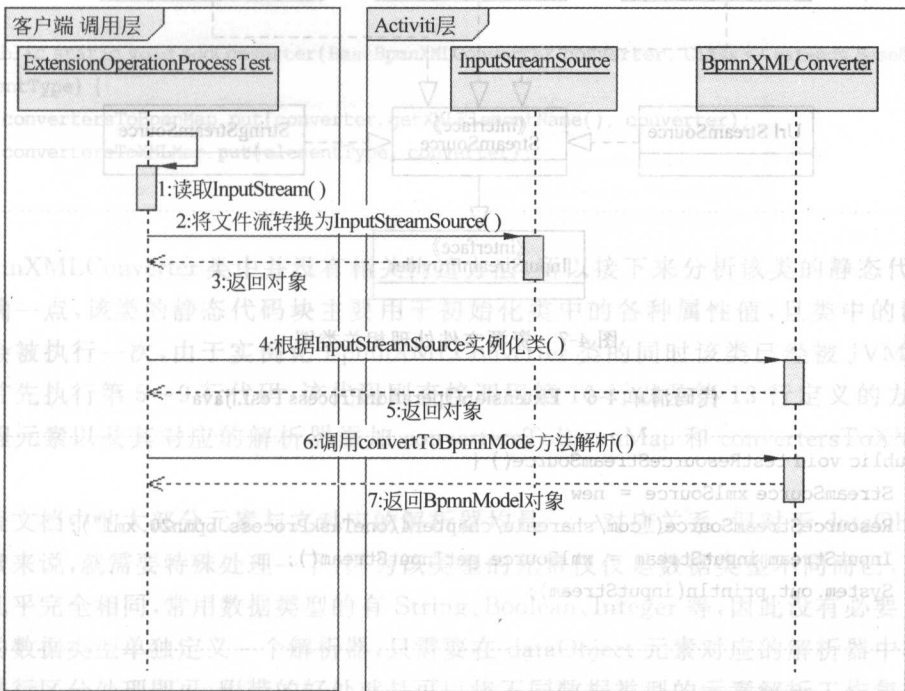


图 4-6 BpmnXMLConverter 类的 convertToBpmnModel 方法时序图

上面的代码中,构造了 StreamSource 实例对象,这样后续进行资源处理时就可以通过 StreamSource 实例对象获取文件流信息,那么 StreamSource 文件流是如何封装的呢?

4.3.2 封装流程文档数据流

StreamSource 接口的定义如代码清单 4-5 所示。

代码清单 4-5 StreamSource.java

```
1 public interface StreamSource extends InputStreamProvider {
2     InputStream getInputStream();
3 }
```

StreamSource 接口继承 InputStreamProvider 接口,该接口只定义了一个方法 getInputStream(),该方法返回 InputStream 实例对象,不同来源的资源文件都有相应的 StreamSource 实现:如 byte 数组(BytesStreamSource)、InputStream 资源(InputStreamSource)、URL 网络资源(UrlStreamSource)、classpath 资源(ResourceStreamSource)等,该类的类图如图 4-7 所示。

在实际项目开发中,资源文件的定位以及数据流的获取需要经常使用,如果不打算自己实现则可以直接通过 Activiti 提供的相关类进行操作,如代码清单 4-6 所示。

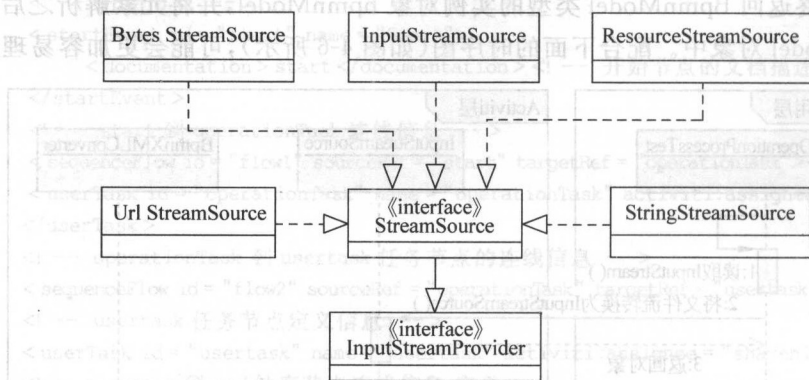


图 4-7 资源文件处理相关类图

代码清单 4-6 ExtensionOperationProcessTest.java

```

1 public void testResourceStreamSource() {
2     StreamSource xmlSource = new
3     ResourceStreamSource("com/shareniu/chapter4/oneTaskProcess.bpmn20.xml");
4     InputStream inputStream = xmlSource.getInputStream();
5     System.out.println(inputStream);
6 }

```

获取到 inputStream 实例对象之后,就可以按照平时的开发方式进行相关实现。StreamSource 接口可以对需要操作的资源文件进行统一处理,其实现原理非常简单,以 ResourceStreamSource 类中的 getInputStream 方法为例,该方法的实现方式便是直接调用 classLoader 提供的底层方法进行操作。

StreamSource 接口定义了 getInputStream 方法,并提供了一系列访问不同资源的实现类,而 BpmnXMLConverter 类中的 convertToBpmnModel 方法需要 InputStreamProvider 类型的参数,这样的设计是典型的策略模式,了解了 StreamSource 类的相关设计原理之后,接下来深入探究 BpmnXMLConverter 类的初始化过程。

4.3.3 初始化元素解析器

代码清单 4-4 实例化了 BpmnXMLConverter 类,实例化该类的同时流程引擎做了什么工作呢? 首先进入 BpmnXMLConverter 类,该类的相关定义如代码清单 4-7 所示。

代码清单 4-7 BpmnXMLConverter.java

```

1 BpmnEdgeParser bpmnEdgeParser = new BpmnEdgeParser();
2 //实例化一系列的解析器
3 protected static Map<String, BaseBpmnXMLConverter> convertersToBpmnMap;
4 Map<Class<? extends BaseElement>, BaseBpmnXMLConverter> convertersToXMLMap;
5 static {
6     ...//省略一系列的解析器添加过程
7     addConverter(new EndEventXMLConverter());
8     addConverter(new ValuedDataObjectXMLConverter(), StringDataObject.class);

```

```
9 }
10 public static void addConverter(BaseBpmnXMLConverter converter) {
11     addConverter(converter, converter.getBpmnElementType());
12 }
13 public static void addConverter(BaseBpmnXMLConverter converter, Class<? extends BaseElement>
    elementType) {
14     convertersToBpmnMap.put(converter.getXMLElementName(), converter);
15     convertersToXMLMap.put(elementType, converter);
16 }
```

BpmnXMLConverter 类中并没有相关构造方法,所以接下来分析该类的静态代码块,首先明确一点,该类的静态代码块主要用于初始化类中的各种属性值,且类中的静态代码块只会被执行一次,由于实例化 BpmnXMLConverter 类的时候该类已经被 JVM 加载,所以会首先执行第 5~9 行代码,该代码则直接调用第 10 行以及第 13 行定义的方法,最终将流程元素以及其对应的解析器添加 convertersToBpmnMap 和 convertersToXMLMap 集合。

流程文档中的大部分元素与之对应的解析器均是一一对应关系,但对于 dataObject 类型的元素来说,就需要特殊处理一下,因为该类型的元素仅仅是数据类型不同而已,其他属性定义几乎完全相同,常用数据类型的有 String、Boolean、Integer 等,因此没有必要为每一种具体的数据类型单独定义一个解析器,只需要在 dataObject 元素对应的解析器中根据数据类型进行区分处理即可,附带的好处就是可以将不同数据类型的元素解析工作集中起来管理,这样也可以控制不同数据类型的元素按照指定的先后顺序进行解析。

注意

ValuedDataObjectXMLConverter 类负责解析 dataObject 元素。

第 5~9 行代码执行完之后,开始实例化第 1~2 行中的各种内置元素解析器,例如 signal 元素的解析器 SignalParser。第 14 行使用 convertersToBpmnMap 集合存储元素以及元素对应的解析器,该集合为 Map 数据结构, key 为 String 类型,存储流程文档中定义的元素名称,对应 converter.getXMLElementName() 方法的返回值(流程文档中元素的名称), value 为元素对应的解析器,例如解析结束事件(endEvent)元素的时候可以直接从 convertersToBpmnMap 集合中查找 key 为 endEvent 的值,这样就可以查询到 EndEventXMLConverter 类。

思考一个问题:为什么使用 Map 数据结构存储,而不是其他数据结构方式存储,例如 List 方式存储,关于这样设计的动机和意图可以参考 4.4.1 节的讲解。

4.3.4 文档转换器功能

了解了 BpmnXMLConverter 类的初始化过程之后,接下来分析该类的功能结构,如图 4-8 所示。

图 4-8 描述了 BpmnXMLConverter 类中的核心方法,下面细化讲解该类中的方法所提供的功能。

(1) convertToBpmnModel 方法:解析流程文档中的元素,最终将元素解析结果封装为

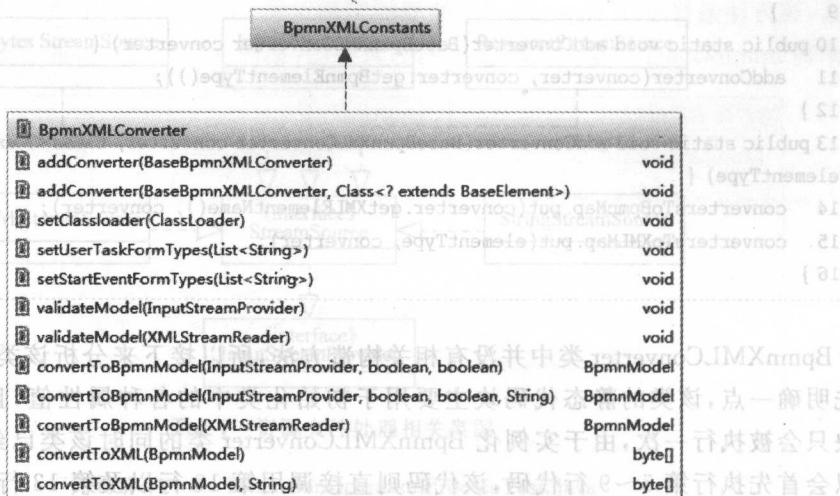


图 4-8 BpmnXMLConverter 类功能结构图

BaseElement 实例对象,该方法最终返回 BpmnModel 实例对象。可以将 BpmnModel 实例对象理解为流程文档解析之后的内存对象,流程文档中所有元素的解析结果均存储在该实例对象中,开发人员可以直接通过该实例对象获取流程文档中定义的所有元素信息。

(2) convertToXML 方法: 将 BpmnModel 实例对象转化为流程文档内容,该方法的操作与 convertToBpmnModel 方法的操作完全相反,convertToBpmnModel 方法则将流程文档内容转化为 BpmnModel 实例对象。

(3) validateModel 方法: 使用 BPMN20. xsd 文件以及该文件所引入的其他 XSD 文件来验证流程文档中定义的元素是否符合其约束。

(4) addConverter 方法: 向 BpmnXMLConverter 类中的 convertersToBpmnMap 和 convertersToXMLMap 集合添加元素解析器,开发人员可以通过该方法添加自定义元素解析器从而替换引擎默认的元素解析器。该方法非常重要。

4.3.5 元素解析环境准备

接下来详细分析 BpmnXMLConverter 类中 convertToBpmnModel 方法的实现逻辑,如代码清单 4-8 所示。

代码清单 4-8 BpmnXMLConverter.java

```

1 BpmnModel convertToBpmnModel (InputStreamProvider inputStreamProvider, boolean validateSchema,
2 boolean enableSafeBpmnXml, String encoding) {
3     XMLInputFactory xif = XMLInputFactory.newInstance();
4     if (xif.isPropertySupported(XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES)) {
5         xif.setProperty(XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES, false);
6     }
7     ...//省略设置 xif 对象中的属性
8     InputStreamReader in = null;
9     try {

```

```

9    in = new InputStreamReader(inputStreamProvider.getInputStream(), encoding);
10   XMLStreamReader xtr = xif.createXMLStreamReader(in);
11   try {
12     if (validateSchema) {
13       if (!enableSafeBpmnXml) {
14         validateModel(inputStreamProvider);
15       } else {
16         validateModel(xtr);
17       }
18     }
19     in = new InputStreamReader(inputStreamProvider.getInputStream(), encoding);
20     xtr = xif.createXMLStreamReader(in);
21   }
22   return convertToBpmnModel(xtr);
23 } finally {
24   if (in != null) {
25     in.close();
26   }
27 }

```

上文详细讲解了 STAX 解析 XML 的处理流程,有了前面的学习基础,相信可以很轻松地掌握 convertToBpmnModel 方法的执行逻辑,下面对该方法的执行逻辑加以总结。

(1) 第 2 行实例化 XMLInputFactory 工厂。
(2) 第 3~6 行为 XMLInputFactory 实例对象添加防护措施,防止外部 DTD 或者 XSD 文件入侵。

(3) 第 9 行根据 inputStreamProvider 对象中的文件流实例化 InputStreamReader 类。

(4) 第 10 行创建 XMLStreamReader 实例对象。

(5) 第 12 行如果开启了 Schema 文件验证,则需要验证流程文档中定义的元素是否符合 XSD 文件约束要求。

(6) 流程文档验证之后,第 18~19 行需要重新打开 InputStreamReader 流并实例化 XMLStreamReader 类,因为 Schema 文件验证完毕之后该流已经被关闭了,因此需要重新打开该流。

(7) 以上所有步骤操作无误之后,第 22 行直接委托 convertToBpmnModel(xtr) 方法解析流程文档元素,配合时序图如图 4-9 所示,可能会更容易理解。

(8) 第 23~26 行关闭文件流。

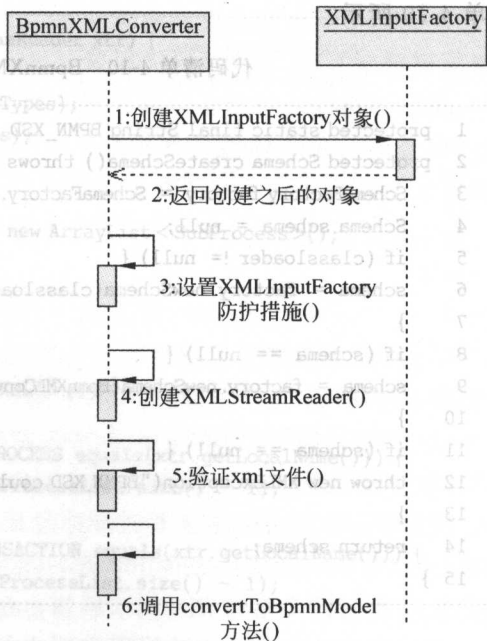


图 4-9 调用 BpmnXMLConverter 类的 convertToBpmnModel 方法时序图

4.3.6 验证流程文档格式

从代码清单 4-8 的处理逻辑可以看出,只有 validateSchema 参数值为 true 的情况下才会开启流程文档元素的验证工作,开启流程文档元素验证之后,根据 enableSafeBpmnXml 参数值执行不同的逻辑,具体实现如代码清单 4-9 所示。

代码清单 4-9 BpmnXMLConverter.java schema 验证 XML

```

1 public void validateModel(InputStreamProvider inputStreamProvider) throws Exception {
2     Schema schema = createSchema();
3     Validator validator = schema.newValidator();
4     validator.validate(new StreamSource(inputStreamProvider.getInputStream()));
5 }
6 //验证是否符合 Bpmn 规范
7 public void validateModel(XMLStreamReader xmlStreamReader) throws Exception {
8     Schema schema = createSchema();
9     Validator validator = schema.newValidator();
10    validator.validate(new StAXSource(xmlStreamReader));
11 }

```

通过分析上面代码的处理逻辑可知不管使用什么方式验证 Schema 文件,首先都会调用 createSchema 方法创建 Schema 实例对象,然后基于该对象获取验证器,最后直接使用验证器进行流程文档的验证工作,唯一的区别就是第 4 行代码使用 StreamSource 实例对象,第 10 行代码使用 StAXSource 实例对象。下面分析 createSchema 方法的执行逻辑,如代码清单 4-10 所示。

代码清单 4-10 BpmnXMLConverter.java 创建 Schema

```

1 protected static final String BPMN_XSD = "org/activiti/impl/bpmn/parser/BPMN20.xsd";
2 protected Schema createSchema() throws SAXException {
3     SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
4     Schema schema = null;
5     if (classloader != null) {
6         schema = factory.newSchema(classloader.getResource(BPMN_XSD));
7     }
8     if (schema == null) {
9         schema = factory.newSchema(BpmnXMLConverter.class.getClassLoader().getResource(BPMN_XSD));
10    }
11    if (schema == null) {
12        throw new XMLException("BPMN XSD could not be found");
13    }
14    return schema;
15 }

```

创建 schema 对象的逻辑非常简单,首先第 3 行获取工厂类 SchemaFactory,然后第 6 行或者第 9 行调用该工厂类中的 newSchema 方法创建 schema 对象,newSchema 方法依赖 BPMN_XSD 资源文件,第 5 行判断当前类的类加载器 classloader 属性值是否存在,如果存

在则直接通过该类加载器获取 BPMN_XSD 文件流,否则第 8 行判断 schema 是否为空,如果不为空第 9 行通过 BpmnXMLConverter 类获取类加载器,然后再通过该类加载器获取 BPMN_XSD 文件流,第 11 行判断 schema 是否为空,如果为空则程序报错。

通过分析上面代码的处理逻辑可以看出 classloader 的使用优先级最高,如果开发人员想要为 classloader 属性赋值,只需要自定义一个文档转换器并继承 BpmnXMLConverter 类,然后为其设置 classloader 属性值即可。

注意

BPMN_XSD 资源文件对应的 XSD 文件位于 activiti-bpmn-converter-5.21.0.jar 包中。

在正式环境建议设置 enableSafeBpmnXml 参数值为 true,这样 Activiti 引擎解析流程文档时会立即验证流程文档中定义的元素是否符合 BPMN20.xsd 文件的约束要求,方便及早发现错误信息。

4.4 元素解析

4.4.1 元素解析入口

代码清单 4-8 中,元素解析的运行环境(加载和验证流程文档信息)准备完毕,开始调用 convertToBpmnModel 方法解析元素,该方法的具体实现如代码清单 4-11 所示。

代码清单 4-11 BpmnXMLConverter.java

```
1 public BpmnModel convertToBpmnModel(XMLStreamReader xtr) {
2     BpmnModel model = new BpmnModel();
3     model.setStartEventFormTypes(startEventFormTypes);
4     model.setUserTaskFormTypes(userTaskFormTypes);
5     try {
6         Process activeProcess = null;
7         List<SubProcess> activeSubProcessList = new ArrayList<SubProcess>();
8         while (xtr.hasNext()) {
9             try {
10                 xtr.next();
11             } catch (Exception e) {
12                 throw new XMLException("Error reading XML", e);
13             }
14             if (xtr.isEndElement() && ELEMENT_SUBPROCESS.equals(xtr.getLocalName())) {
15                 activeSubProcessList.remove(activeSubProcessList.size() - 1);
16             }
17             if (xtr.isEndElement() && ELEMENT_TRANSACTION.equals(xtr.getLocalName())) {
18                 activeSubProcessList.remove(activeSubProcessList.size() - 1);
19             }
20             if (xtr.isStartElement() == false) {
21                 continue;
22             }
23             //解析 definitions 元素
```

```

24 if (ELEMENT_DEFINITIONS.equals(xtr.getLocalName())) {
25     definitionsParser.parse(xtr, model);
26 }
27 //省略 resource、signal、message、error、import、itemDefinition、dataStore、interface、
28 ioSpecification、participant、messageFlow
29 else if (ELEMENT_PROCESS.equals(xtr.getLocalName())) {
30     //解析 process 元素
31     Process process = processParser.parse(xtr, model);
32     if (process != null) {
33         activeProcess = process;
34     }
35 //省略 potentialStarter、documentation、textAnnotation、association、extensionElements、
36 subProcess、transaction、BPMNShape、BPMNEdge
37 } else {
38     if (!activeSubProcessList.isEmpty() &&
39         ELEMENT_MULTIINSTANCE.equalsIgnoreCase(xtr.getLocalName())) {
40         multiInstanceParser.parseChildElement(xtr,
41             activeSubProcessList.get(activeSubProcessList.size() - 1), model);
42     } else if (convertersToBpmnMap.containsKey(xtr.getLocalName())) {
43         if (activeProcess != null) {
44             BaseBpmnXMLConverter converter = convertersToBpmnMap.get(xtr.getLocalName());
45             converter.convertToBpmnModel(xtr, model, activeProcess, activeSubProcessList);
46         }
47         for (Process process : model.getProcesses()) {
48             for (Pool pool : model.getPools()) {
49                 if (process.getId().equals(pool.getProcessRef())) {
50                     pool.setExecutable(process.isExecutable());
51                 }
52                 processFlowElements(process.getFlowElements(), process);
53             } catch (XMLException e) {
54                 throw e;
55             } catch (Exception e) {
56                 throw new XMLException("Error processing BPMN document", e);
57             }
58         }
59     }

```

以上是流程文档中元素解析的全过程,该方法内部使用 XMLStreamReader 迭代器遍历元素,其解析处理步骤总结如下。

- (1) 第 2 行实例化 BpmnModel 类,该类负责存储所有元素解析之后的结果。
- (2) 解析流程文档命名空间。

因为 STAX 解析流程文档的顺序是按照流程文档中元素定义的先后顺序自上而下解析的,所以首先解析父元素 definitions,该元素对应的解析器为 DefinitionsParser,该元素可以定义一系列的命名空间 URI,形如< definitions xmlns:activiti="http://activiti.org/bpmn"></definitions>,因为流程文档中的元素名称是由开发者定义的,为了避免命名冲突,需要引入命名空间对元素加以区分。

(3) 解析外围元素。

什么是外围元素呢？首先需要搞清楚这个概念，平时定义的流程元素，例如开始节点等元素均作为 process 元素的子元素存在，这里所说的外围元素指的是作为 definitions 元素的子元素同时还作为 process 元素的兄弟节点存在，形如消息元素 `<definitions><message id="newInvoice"></message></definitions>`，最常见的外围元素有 message、signal 等，流程文档中定义的大部分外围元素是没有先后顺序之分的，既可以在 process 元素之上，也可以在 process 元素之下，因为外围元素种类不多而且不容易变动，所以该类型的元素解析器均在当前类中进行实例化，有关外围元素的解析处理逻辑可以查看对应的解析器。例如消息元素的解析：`messageFlowParser.parse(xtr, model)`，该 parse 方法需要如下两个输入参数：① xtr 参数：程序可以根据该参数值从流程文档中解析元素的属性值；② model 参数：BpmnModel 实例对象，元素解析完毕，可以直接将元解析结果存储到该元素对应的属性承载类实例对象中，然后再将其添加到 BpmnModel 实例对象中。因为外围元素的种类不多，平时开发中也不经常使用，为了减少风险，增加可控度，外围元素的定义以及解析不建议修改和扩展。

(4) 第 38~57 行开始解析元素，主要解析事件、网关、活动等元素信息。

第 43 行首先判断 activeProcess 对象是否为空，如果该对象为空，表示流程文档中没有定义子元素，所以就不需要进行解析；如果该对象不为空，则第 44 行首先根据 `xtr.getLocalName()` 方法获取元素名称，然后根据元素名称从 `convertersToBpmnMap` 集合中查找该元素对应的解析器，例如 `xtr.getLocalName()` 值为 `userTask`，则对应的解析器为 `UserTaskXMLConverter`，从这里的处理逻辑就可以看出 `convertersToBpmnMap` 集合选用 Map 数据结构的好处，如果使用 List 数据结构，势必存在如下两个问题：① 需要循环遍历解析器集合才能查找到适配当前元素的解析器；② 客户端向该集合添加元素解析器时，可能会造成同一个元素的解析器有多个；如果同一个元素对应多个解析器，那么引擎该如何选择解析器，即使多个解析器都可以正常使用，引擎又该以哪一个解析器的解析结果为准？如果使用 Map 数据结构就不会出现以上问题，客户端可以直接根据 key 值（元素名称）查找元素对应的解析器，而且相同的 key 值只能存储一个，所以如果想要使用自定义元素解析器，只需要根据 key 值覆盖引擎默认的元素解析器即可。

(5) 解析通用元素。

流程文档中通用元素的种类非常多，例如文档元素“documentation”、扩展元素“extensionElements”等，这些元素通常作为流程定义三大要素的子元素存在，试想一下，如果每个元素都在自己的解析处理逻辑中对通用元素进行解析，势必会造成相同的解析代码分散在各个模块中，无形之中增加了项目维护的复杂度，如果通用元素的解析规则变了，则需要修改每个模块中对应的解析逻辑，工作量非常大，如果将通用元素的解析功能抽离出来进行统一管理维护，则以上这些问题就不会出现。虽然暂时还没有看到具体元素的解析过程，但是已经看到了大量类似 `xxx.parse` 方法的调用，因为元素解析是基于 STAX 迭代器方式，所以首先会获取元素类型，然后再根据元素类型委托不同的解析器进行解析，上文细讲解过这样设计的好处，相信结合该方法的处理逻辑对开闭原则的掌握会更加深入。

由于 Activiti 中的元素类型非常多，从而导致元素解析器非常庞大，因为元素的解析处理步骤大体类似，每个都进行讲解，工作量极大，而且可能会有事倍功半的效果，所以接下来

重点讲解通用元素以及流程元素(如连线元素)的解析处理逻辑,其他元素的解析逻辑可以参考该案例自行学习。

4.4.2 解析根元素

根元素 definitions 对应的解析器为 DefinitionsParser,根元素中可以定义多个流程元素 process(建议每个流程文档只定义一个 process 元素,这样可以减少开发过程中的维护成本),根元素 definitions 的定义以及解析过程如代码清单 4-12 所示。

代码清单 4-12 DefinitionsParser.java 和根元素的定义

1 根元素的定义:

```
2 <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
```

```
3 xmlns:activiti="http://activiti.org/bpmn" targetNamespace="shareniu" id="shareniu"
name="shareniu" exporter="shareniu" exporterVersion="1">
```

4 DefinitionsParser.java 如下:

```
5 protected static final List<ExtensionAttribute> defaultAttributes = Arrays.asList(
6     new ExtensionAttribute("typeLanguage"), new ExtensionAttribute("expressionLanguage"),
7     new ExtensionAttribute("targetNamespace")
8 );
9 public void parse(XMLStreamReader xtr, BpmnModel model) throws Exception {
10     //获取 targetNamespace 命名空间,对应值为"shareniu"
11     model.setTargetNamespace(xtr.getAttributeValue(null, TARGET_NAMESPACE_ATTRIBUTE));
12     for (int i = 0; i < xtr.getNamespaceCount(); i++) { //获取命名空间形如 xmlns:activiti
13         String prefix = xtr.getNamespacePrefix(i); //形如 activiti
14         if (StringUtils.isEmpty(prefix)) {
15             //xtr.getNamespaceURI(i)值对应"http://activiti.org/bpmn"
16             model.addNamespace(prefix, xtr.getNamespaceURI(i));
17         }
18         for (int i = 0; i < xtr.getAttributeCount(); i++) { //获取属性值形如以上的 id,name 等
19             ExtensionAttribute extensionAttribute = new ExtensionAttribute(); //扩展属性承载类
20             extensionAttribute.setName(xtr.getAttributeLocalName(i));
21             extensionAttribute.setValue(xtr.getAttributeValue(i));
22             if (StringUtils.isEmpty(xtr.getAttributeNamespace(i))) {
23                 extensionAttribute.setNamespace(xtr.getAttributeNamespace(i)); //存在命名空间则添加
24             }
25             if (StringUtils.isEmpty(xtr.getAttributePrefix(i))) { //不为空添加
26                 extensionAttribute.setNamespacePrefix(xtr.getAttributePrefix(i));
27             }
28             if (!BpmnXMLUtil.isBlacklisted(extensionAttribute, defaultAttributes)) {
29                 model.addDefinitionsAttribute(extensionAttribute);
30             }
31 }
```

根元素的解析处理流程总结如下。

- (1) 第 11 行获取根元素的 targetNamespace 属性值并将其设置到 model 对象中。
- (2) 第 13~17 行遍历所有命名空间的 URI,包括获取 URI 的前缀以及值,最终第 16

行将其添加到 model 对象中。

同(3)第 18~30 行进行属性解析。

(4) 第 28 行验证黑名单列表,如果解析的属性不在黑名单列表中,则将其作为扩展属性进行处理,并通过第 29 行操作将其添加到 model 对象中。

注意

definitions 元素至少需要包含 xmlns 和 targetNamespace 两个属性声明, targetNamespace 可以是任意值,该属性可以用来对流程定义模板进行分类(项目开发中的一个小技巧)。

上述代码中进行了黑名单列表验证,接下来详细分析其处理逻辑。对于 definitions 元素来说,默认的黑名单属性集合 defaultAttributes 在第 5~8 行中进行初始化,该集合中的元素有如下 3 个: typeLanguage、expressionLanguage、targetNamespace,其处理流程如下。

- 遍历 definitions 元素中的所有属性。对于 definitions 元素来说,除了定义黑名单中的属性,还分别定义了 exporterVersion、id、name、exporter 4 个属性,因此以上所述的 7 个属性都会进行遍历。
- 封装属性信息。ExtensionAttribute 类承载了定义的属性名称 name、属性值 value、命名空间 namespace、命名空间前缀 namespacePrefix 信息。
- definitions 元素的扩展属性判断。

BpmnXMLUtil.isBlacklisted() 方法的处理逻辑如代码清单 4-13 所示。

代码清单 4-13 BpmnXMLUtil.java

```

1 public static boolean isBlacklisted(ExtensionAttribute attribute, List<ExtensionAttribute>...
  blackLists) {
2     if (blackLists != null) {
3         for (List<ExtensionAttribute> blackList : blackLists) {
4             for (ExtensionAttribute blackAttribute : blackList) {
5                 if (blackAttribute.getName().equals(attribute.getName())) {
6                     if (blackAttribute.getNamespace() != null && attribute.getNamespace() != null
7                         && blackAttribute.getNamespace().equals(attribute.getNamespace()))
8                         return true;
9                     if (blackAttribute.getNamespace() == null && attribute.getNamespace() == null)
10                        return true;
11                 }
12             }
13         }
14     }
15     return false;
16 }

```

该方法的处理逻辑如下。

- 第 2 行如果 blackLists 集合为空,则返回 false;如果 blackLists 集合不为空,第 3 行循环遍历该集合,第 4 行循环遍历 blackList,第 5 行如果 blackAttribute 对象中的

name 与 attribute 对象的 name 相同,则第 6~7 行开始继续比对两个对象的命名空间值是否相等,如果相等第 8 行返回 true,第 9~10 行如果两个对象的命名空间均为空也返回 true。

- isBlacklisted 方法执行完毕后直接返回处理结果。

(5) 经过 isBlacklisted 方法处理之后,如果存在扩展属性,则通过 model.addAttribute 方法将其添加到 model 对象中,方便后续获取,addDefinitionsAttribute 方法内部使用 Map 数据结构存储扩展信息,该方法的实现相对来说比较简单,本书不过多讲解。看到这里可能会有疑问:哪些元素支持扩展属性呢?目前,Activiti 中支持该特性的元素有 3 个:根元素 definitions、任务节点 userTask 和流程元素 process。

4.4.3 流程内元素解析入口

流程元素 process 的解析过程可以参考 ProcessParser 类的相关实现,接下来暂且将关注点放到 process 元素的子元素解析过程中,在代码清单 4-11 中,第 44 行根据元素名称从集合中查找到该元素对应的解析器,因为集合 convertersToBpmnMap 在 BpmnXMLConverter 类加载的时候已经初始化,关于这一点上文也详细讲解过,查找到元素对应的解析器之后,直接委托解析器的基类 BaseBpmnXMLConverter 中的 convertToBpmnModel 方法解析元素,该方法的相关定义如代码清单 4-14 所示。

代码清单 4-14 BaseBpmnXMLConverter.java

```

1 public void convertToBpmnModel(XMLStreamReader xtr, BpmnModel model, Process activeProcess,
2 List<SubProcess> activeSubProcessList) throws Exception {
3     String elementId = xtr.getAttributeValue(null, ATTRIBUTE_ID); //获取 id 值
4     String elementName = xtr.getAttributeValue(null, ATTRIBUTE_NAME); //获取 name 值
5     boolean async = parseAsync(xtr); //获取 activiti:async 中的值
6     boolean notExclusive = parseNotExclusive(xtr); //获取 activiti:exclusive 中的值
7     String defaultFlow = xtr.getAttributeValue(null, ATTRIBUTE_DEFAULT); //获取 default 中的值
8     boolean isForCompensation = parseForCompensation(xtr);
9     //调用具体的解析器中的 convertXMLToElement 方法进行解析
10    BaseElement parsedElement = convertXMLToElement(xtr, model);
11    if (parsedElement instanceof Artifact) {
12        Artifact currentArtifact = (Artifact) parsedElement;
13        currentArtifact.setId(elementId);
14        if (!activeSubProcessList.isEmpty()) {
15            activeSubProcessList.get(activeSubProcessList.size() - 1).addArtifact(currentArtifact);
16        } else {
17            activeProcess.addArtifact(currentArtifact);
18        }
19        if (parsedElement instanceof FlowElement) { //流程三大要素中的元素基本都是该类型
20            FlowElement currentFlowElement = (FlowElement) parsedElement;
21            currentFlowElement.setId(elementId); //设置 id
22            currentFlowElement.setName(elementName); //设置 name
23            if (currentFlowElement instanceof FlowNode) {
24                FlowNode flowNode = (FlowNode) currentFlowElement;

```

```

25 flowNode.setAsynchronous(async); //是否异步
26 flowNode.setNotExclusive(notExclusive); //是否排它
27 if (currentFlowElement instanceof Activity) { //活动节点
28     Activity activity = (Activity) currentFlowElement;
29     activity.setForCompensation(isForCompensation);
30     if (StringUtils.isEmpty(defaultFlow)) {
31         activity.setDefaultFlow(defaultFlow); //默认连线
32     }
33     if (currentFlowElement instanceof Gateway) { //网关
34         Gateway gateway = (Gateway) currentFlowElement;
35         if (StringUtils.isEmpty(defaultFlow)) {
36             gateway.setDefaultFlow(defaultFlow);
37         }
38         if (currentFlowElement instanceof DataObject) {
39             if (!activeSubProcessList.isEmpty()) {
40                 activeSubProcessList.get(activeSubProcessList.size()
41                     - 1).getDataObjects().add((ValuedDataObject) parsedElement);
42             } else {
43                 activeProcess.getDataObjects().add((ValuedDataObject) parsedElement);
44             }
45             if (!activeSubProcessList.isEmpty()) { //解析元素属于子流程
46                 activeSubProcessList.get(activeSubProcessList.size() - 1).addFlowElement(currentFlowElement);
47             } else {
48                 activeProcess.addFlowElement(currentFlowElement);
49             }
50         }
51     }
52 }

```

以上便是 process 元素的子元素的解析全过程,在解析元素的所有属性之前,Activiti 做了全局功能架构,主要包括如下内容。

(1) 第 3~8 行解析公共属性值,常用的元素属性包括但不限于 id、name、async、exclusive、default、isForCompensation 几种。

(2) 元素属性解析。

元素公共属性解析完毕,第 10 行开始调用 convertXMLToElement(xtr, model) 方法执行具体元素的解析工作,因为 convertXMLToElement(xtr, model) 方法是当前类 BaseBpmnXMLConverter 中定义的一个抽象方法,所以最终会调用具体的元素解析器完成元素解析工作,该方法解析完毕之后返回 BaseElement 实例对象。换言之,元素以及属性解析完毕之后会将其解析结果封装为 BaseElement 实例对象。

(3) 第 11~36 行后置赋值操作。

根据元素对应的属性承载类的类型进行赋值,该操作主要区分网关和活动(引用流程、子流程以及所有的 Task 节点)两大要素,所有的节点均需要添加 id、name 值,如果元素是网关类型则需要填充 defaultFlow、asynchronous、notExclusive 属性值,“活动”类型则需要填充 asynchronous、notExclusive、forCompensation、defaultFlow 属性值。

(4) 第 38~43 行如果元素类型为 DataObject,则需要判断 activeSubProcessList 参数值是否为空,如果不为空,则将其作为子流程中的元素进行添加,否则添加到 activeProcess 对象中。

(5) 第 44~47 行判断当前正在解析元素的父级元素,如果 activeSubProcessList 参数

值不为空,则将其添加到 activeSubProcessList 对象中,否则添加到 activeProcess 对象中。

4.4.4 解析连线

上面提到 BaseBpmnXMLConverter 类中的 convertXMLToElement(xtr, model) 是抽象方法,该方法需要交给具体的子类去实现,为何这样设计? 因为流程文档中的每个元素都对应一个解析器以实现自身属性解析功能,不同的解析器内部实现不同,但都在父类的 convertXMLToElement 方法中进行统一调用,该方法作为通用模板方法存在,由于 BpmnXMLConstants 接口定义了流程文档中所有的元素以及属性字段,为了便于统一管理,所有的元素解析器均直接或者间接地实现该接口。

下面详细讲解 sequenceFlow 元素的解析处理流程,该元素的解析器 SequenceFlowXMLConverter 中 convertXMLToElement 方法的相关实现如代码清单 4-15 所示。

代码清单 4-15 SequenceFlowXMLConverter.java convertXMLToElement 方法

```
1 protected BaseElement convertXMLToElement(XMLStreamReader xtr, BpmnModel model) throws
Exception{
2     SequenceFlow sequenceFlow = new SequenceFlow();
3     BpmnXMLUtil.addXMLLocation(sequenceFlow, xtr);
4     sequenceFlow.setSourceRef(xtr.getAttributeValue(null, ATTRIBUTE_FLOW_SOURCE_REF));
5     sequenceFlow.setTargetRef(xtr.getAttributeValue(null, ATTRIBUTE_FLOW_TARGET_REF));
6     sequenceFlow.setName(xtr.getAttributeValue(null, ATTRIBUTE_NAME));
7     sequenceFlow.setSkipExpression(xtr.getAttributeValue(null, ATTRIBUTE_FLOW_SKIP_EXPRESSION));
8     parseChildElements(getXMLElementName(), sequenceFlow, model, xtr);
9     return sequenceFlow;
10 }
```

convertXMLToElement 方法的处理逻辑如下所示。

(1) 第 2 行实例化 sequenceFlow 元素对应的属性承载类,该类承载了连线元素所有属性值的获取和存储工作,连线元素解析完毕,所有的属性值都会封装到 SequenceFlow 实例对象中。

(2) 第 3 行获取元素在 XML 中的坐标信息。

获取元素在流程文档中的行号和列号是非常必要的,这样做的好处就是当元素解析失败时,会将该元素在流程文档中的行号和列号暴露给客户端,方便客户端定位和排查问题。

(3) 第 4~7 行解析属性信息。

对于 sequenceFlow 元素来说,常用的属性有 sourceRef、targetRef、name、skipExpression,将这些属性信息获取并填充到 sequenceFlow 对象中。

(4) 第 8 行开始解析当前元素的子元素。

调用 parseChildElements 方法解析 sequenceFlow 元素中的子元素,对于 sequenceFlow 元素来说,常用的子元素有文档元素 documentation、扩展元素 extensionElements(包括监听器以及用户自定义元素)。

4.4.5 获取元素坐标

在上述代码中,第3行委托 BpmnXMLUtil.addXMLLocation()方法获取连线元素在XML文档中的行号和列号信息,具体实现如代码清单4-16所示。

代码清单4-16 BpmnXMLUtil.java 获取元素文本结束位置的行号和列号

```
1 public static void addXMLLocation(BaseElement element, XMLStreamReader xtr) {  
2     Location location = xtr.getLocation();  
3     element.setXmlLineNumber(location.getLineNumber());  
4     element.setXmlColumnNumber(location.getColumnNumber());  
5 }
```

首先根据 xtr 对象获取 Location 实例对象,然后调用 location 对象的 getLineNumber 和 getColumnNumber 方法分别获取到行号和列号,获取完毕赋值到 BaseElement 实例对象中(对于连线来说该实例对象为 sequenceFlow)。

注意

BpmnXMLUtil 类被 JVM 加载时会触发该类中的静态代码块。

4.5 子元素解析

4.5.1 初始化子元素解析器

在上述代码中,BpmnXMLUtil.addXMLLocation()方法用于获取元素坐标信息,因为 addXMLLocation()方法为 BpmnXMLUtil 类中的静态方法,所以该方法被调用的同时会触发该类的静态代码块,如代码清单4-17所示。

代码清单4-17 BpmnXMLUtil.java 子元素解析器初始化

```
1 private static Map<String, BaseChildElementParser> genericChildParserMap;  
2 static {  
3     addGenericParser(new ActivitiEventListenerParser());  
4     ...//省略一系列的子元素解析器  
5     addGenericParser(new ActivitiMapExceptionParser());  
6 }  
7 private static void addGenericParser(BaseChildElementParser parser) {  
8     genericChildParserMap.put(parser.getElementName(), parser);  
9 }
```

该静态代码块会加载所有的通用子元素解析器,并且只会加载一次,这里所说的通用子元素均指流程元素中的子元素信息,例如所有的流程元素都有文档描述元素 "documentation"、"extensionElements" 元素(可以定义执行监听器等)、条件表达式 "conditionExpression"、多实例 "multiInstanceLoopCharacteristics" 等,因为这些子元素比较

通用,所以将其单独抽取出来进行定义和解析,降低程序维护成本,所有的子元素解析器均需继承 BaseChildElementParser 类。上述代码中涉及了 addGenericParser 方法的调用,该方法是 BpmnXMLUtil 类的有私有静态方法,通过这里的设计可以看出,客户端不能通过该方法添加自定义子元素解析器,也不能入侵和干预子元素的解析工作,有没有其他办法可以替换默认子元素解析器呢?带着这个疑问,分析子元素的解析过程,期望从中找到答案。

注意

genericChildParserMap 是 Map 数据结构, key 为流程文档中对应的子元素名称, value 为子元素对应的解析器。

4.5.2 解析子元素

在代码清单 4-15 中,调用了 parseChildElements 方法来解析当前元素的子元素,但是 SequenceFlowXMLConverter 类中并没有该方法的实现,既然该类中没有提供实现,那么其父类肯定提供了默认实现,所以找到当前类的父类 BaseBpmnXMLConverter,相关实现如代码清单 4-18 所示。

代码清单 4-18 BaseBpmnXMLConverter.java

```

1 void parseChildElements(String elementName, BaseElement parentElement, BpmnModel model,
XMLStreamReader xtr) {
2     parseChildElements(elementName, parentElement, null, model, xtr);
3 }
4 protected void parseChildElements (String elementName, BaseElement parentElement, Map
<String, BaseChildElementParser > additionalParsers, BpmnModel model, XMLStreamReader xtr)
throws Exception {
5     Map<String, BaseChildElementParser> childParsers = new HashMap();
6     if (additionalParsers != null) {
7         childParsers.putAll(additionalParsers);
8     }
9     BpmnXMLUtil.parseChildElements(elementName, parentElement, xtr, childParsers, model);
10 }

```

在上述代码中,第 2 行直接委托第 4 行定义的方法解析子元素,并且设置 additionalParsers 参数值为 null,该参数值为客户端自定义的子元素解析器集合,通过该步骤可以看出程序默认不加载用户自定义的子元素解析器,如果客户端需要添加自定义子元素解析器可以直接设置 additionalParsers 参数值。

第 6 行判断 additionalParsers 集合是否为空,如果不为空,则第 7 行将用户自定义的子元素解析器集合添加到 childParsers 集合中,然后第 9 行委托 BpmnXMLUtil 类中的 parseChildElements 方法解析子元素,parseChildElements 方法的相关实现如代码清单 4-19 所示。

代码清单 4-19 BpmnXMLUtil.java 解析子元素

```

1 public static void parseChildElements (String elementName, BaseElement parentElement,

```

```

XMLStreamReader xtr, Map<String, BaseChildElementParser> childParsers, BpmnModel model){
2  Map<String, BaseChildElementParser> localParserMap = new HashMap(genericChildParserMap);
3      if (childParsers != null) {
4          localParserMap.putAll(childParsers);
5      }
6      boolean inExtensionElements = false;
7      boolean readyWithChildElements = false;
8      while (readyWithChildElements == false && xtr.hasNext()) {
9          xtr.next(); //移动读取器游标
10         if (xtr.isStartElement()) { //开始元素
11             if ("extensionElements".equals(xtr.getLocalName())) {
12                 inExtensionElements = true; //如果是扩展元素设置该变量值为 true
13             } else if (localParserMap.containsKey(xtr.getLocalName())) { //从通用子元素集合中获取查找
14                 BaseChildElementParser childParser = localParserMap.get(xtr.getLocalName());
15                 //如果当前元素是 extensionElements 元素的子元素
16                 if (inExtensionElements && !childParser.accepts(parentElement)) {
17                     //开始解析用户自定义扩展元素
18                     ExtensionElement extensionElement = BpmnXMLUtil.parseExtensionElement(xtr);
19                     //将解析之后的元素对象添加到父级元素中
20                     parentElement.addExtensionElement(extensionElement);
21                     continue;
22                 }
23                 //开始解析通用子元素
24                 localParserMap.get(xtr.getLocalName()).parseChildElement(xtr, parentElement, model);
25             } else if (inExtensionElements) { //用户自定义扩展元素
26                 ExtensionElement extensionElement = BpmnXMLUtil.parseExtensionElement(xtr);
27                 parentElement.addExtensionElement(extensionElement);
28             }
29             } else if (xtr.isEndElement()) {
30                 if ("extensionElements".equals(xtr.getLocalName())) {
31                     inExtensionElements = false;
32                 } else if (elementName.equalsIgnoreCase(xtr.getLocalName())) {
33                     readyWithChildElements = true;
34                 }
35             }
36         }
37     }

```

子元素的解析流程可以大致总结如下。

(1) 第2~5行初始化子元素解析器集合 localParserMap。

该集合的初始化工作分如下两步：

① 获取内置子元素解析器集合 genericChildParserMap 的值并将其添加到 localParserMap 集合中, genericChildParserMap 集合的初始化过程可以参考该类的静态代码块；

② 获取 childParsers 参数值, 如果该参数值不为空, 则将该值添加到子元素解析器集合 localParserMap 中, 因为 localParserMap 是 Map 数据结构, 所以用户自定义的子元素解析器可以覆盖 Activiti 默认的子元素解析器, 该参数值的有无非常重要, 直接影响子元素解析

时所需要使用的解析器。

(2) 扩展元素判断。

第 16 行 `childParser.accepts(parentElement)` 方法的处理至关重要,如果父类元素不能识别文档中定义的子元素,那么引擎就认为这个元素为扩展元素并在第 18~20 行对其进行处理。

(3) 解析扩展元素。

子元素的解析可以分为通用子元素解析(如“documentation”元素)和用户自定义扩展元素解析,其处理逻辑为:如果经过判断发现子元素为通用子元素,第 24 行根据子元素的名称获取子元素解析器进行子元素的解析工作;如果为用户自定义扩展元素则直接委托 `BpmnXMLUtil.parseExtensionElement(xtr)` 方法进行解析,该过程可以参考第 5.1 节的讲解。不管解析何种类型的子元素,均需要将元素解析之后的结果添加到父元素 `parentElement` 中。

看到这里可能有些疑问: `parentElement` 是哪一个对象呢? 其实也不难理解,不妨换一个思路思考这个问题, `parseChildElements` 方法是由哪一个对象调用的呢? 很显然该方法由具体解析类的实例对象进行调用,比如现在开始解析任务节点,则任务节点的解析类 `UserTaskXMLConverter` 解析任务节点时会调用 `parseChildElements` 方法进行子元素的解析工作,这时 `parentElement` 对象就对应 `UserTaskXMLConverter` 实例对象。

注意

`childParser.accepts()` 方法的处理过程,可以参考 `FieldExtensionParser` 或者 `FormPropertyParser` 类的实现。

4.5.3 解析扩展元素

下面以 `process` 元素的执行监听器为例,详细分析扩展元素的解析过程,在讲解执行监听器的解析之前,先了解监听器是如何定义的,如代码清单 4-20 所示。

代码清单 4-20 `process` 元素定义监听器和 `SharenIUListener.java`

```

1  流程文档定义如下:
2  <process id="myProcess" name="sharenIUProcess" isExecutable="true">
3    <extensionElements>
4      <activiti:executionListener event="start"
5        class="com.sharenIU.chapter4.SharenIUListener"></activiti:executionListener>
6      <activiti:field name="sharenIU">
7        <activiti:string><![CDATA[sharenIUName]]></activiti:string>
8      </activiti:field>
9    </extensionElements>
10 </process>
11 SharenIUTaskListener.java
12 public class SharenIUListener implements ExecutionListener {
13   Expression sharenIU;           //对应<activiti:field name="sharenIU">中的 name 值
14   private static final long serialVersionUID = 1L;
15   public void notify(DelegateExecution de) {

```

```

16  shareniu.getExpressionText(); //获取的值为<activiti:string>中的值,最终为 shareniuName
17  }
18  }

```

流程文档中的三大要素都可以定义执行监听器和任务监听器(仅限于在任务节点中进行定义)。监听器通常作为扩展元素 `extensionElements` 的子元素进行定义,因为监听器可以很方便的让客户使用和扩展,所以本节详细讲解扩展元素的解析处理流程。执行监听器以及任务监听器的解析器分别为 `ExecutionListenerParser` 和 `TaskListenerParser`,两者均继承了 `BaseChildElementParser` 类,并在父类中进行统一调度。既然监听器通常作为 `extensionElements` 的子元素存在,所以首先找到 `extensionElements` 元素的解析器 `ExtensionElementsParser`,该类的相关定义如代码清单 4-21 所示。

代码清单 4-21 `ExtensionElementsParser.java` 解析监听器元素

```

1  public void parse(XMLStreamReader xtr, List < SubProcess > activeSubProcessList, Process
activeProcess, BpmnModel model) {
2      BaseElement parentElement = null;
3      if (!activeSubProcessList.isEmpty()) {
4          parentElement = activeSubProcessList.get(activeSubProcessList.size() - 1);
5      } else {
6          parentElement = activeProcess;
7      }
8      boolean readyWithChildElements = false;
9      while (readyWithChildElements == false && xtr.hasNext()) {
10         xtr.next();
11         if (xtr.isStartElement()) {
12             if ("executionListener".equals(xtr.getLocalName())) {
13                 new ExecutionListenerParser().parseChildElement(xtr, parentElement, model);
14             } else if ("eventListener".equals(xtr.getLocalName())) {
15                 new ActivitiEventListenerParser().parseChildElement(xtr, parentElement, model);
16             } else if ("potentialStarter".equals(xtr.getLocalName())) {
17                 new PotentialStarterParser().parse(xtr, activeProcess);
18             } else {
19                 ExtensionElement extensionElement = BpmnXMLUtil.parseExtensionElement(xtr);
20                 parentElement.addExtensionElement(extensionElement);
21             }
22         } else if (xtr.isEndElement()) {
23             if ("extensionElements".equals(xtr.getLocalName())) {
24                 readyWithChildElements = true;
25             }
26         }
27     }
28 }

```

`ExtensionElementsParser` 类中 `parse` 方法的处理逻辑如下:

(1) 第 3~6 行确定父级元素的类型,在解析 `extensionElements` 元素之前,首先要确定 `extensionElements` 的父级元素类型。

(2) 第 11 行开始根据 `extensionElements` 元素中的子元素类型执行不同的逻辑, 如果子元素为 `executionListener` 类型(执行监听器)则执行第 13 行代码; 如果子元素为 `eventListener` 类型(事件监听器)则执行第 15 行代码, 事件监听器的配置形如代码清单 4-22 第 2~4 行, 如果子元素为 `potentialStarter` 类型(流程启动器)则执行第 17 行代码, 流程启动器的配置参考代码清单 4-22 中第 6~16 行, 如果子元素不是以上三种类型中的任意一个则执行 19 行代码解析用户自定义元素。

代码清单 4-22 流程事件监听器以及流程启动人的配置

```

1  事件监听器配置:
2  <extensionElements>
3    <activiti:eventListener events = "ENTITY_CREATED" class = "${shareniu}" />
4  </extensionElements>
5  流程启动器配置:
6  <process id = "potentialStarter">
7    <extensionElements>
8      <activiti:potentialStarter>
9        <resourceAssignmentExpression>
10         <formalExpression>shareniu1,group(shareniuGroup2),user(shareniu)</formalExp-
11         ression>
12       </resourceAssignmentExpression>
13     </activiti:potentialStarter>
14   </extensionElements>
15 </process>
16 <process activiti:candidateStarterUsers = "shareniu" activiti:candidateStarterGroups = "sg" >

```

在上述代码中, 第 10 行 `user(shareniu)` 是引用了用户 `shareniu`, `group(shareniuGroup2)` 引用了组 `shareniuGroup2`, 如果定义时没有显式设置, 例如 `shareniu1`, 则引擎默认会将其作为组进行处理。关于 `potentialStarter` 元素的解析处理逻辑可以参考 `PotentialStarterParser` 类中的 `parse` 方法, 对于流程启动人的配置也可以使用 `process` 元素中的属性进行配置如第 15 行代码所示, 其中 `candidateStarterUsers` 用来配置启动器, `candidateStarterGroups` 用来配置组, 如果存在多个人或组则用“,”进行分割。

任务监听器同样也是作为扩展元素 `extensionElements` 的子元素存在, 但是这里并没有发现对任务监听器的处理踪迹, 其实不难理解, 因为流程三大要素中的节点都可以使用执行监听器, 而任务监听器只可以在任务节点进行定义和使用, 因此任务监听器的解析工作只需要在任务节点解析时处理即可, 相关实现可以查看任务节点的解析器 `UserTaskXMLConverter`。接下来分析 `process` 元素中执行监听器的处理逻辑, 相关实现如代码清单 4-23 所示。

代码清单 4-23 ActivitiListenerParser.java

```

1  public void parseChildElement(XMLStreamReader xtr, BaseElement parentElement, BpmnModel
2  model){
3    ActivitiListener listener = new ActivitiListener(); //实例化监听器承载类
4    BpmnXMLUtil.addXMLLocation(listener, xtr); //添加坐标信息
5    if (StringUtils.isEmpty(xtr.getAttributeValue(null, "class"))){ //class 方式创建

```

```

5 listener.setImplementation(xtr.getAttributeValue(null, "class")); //获取属性
6 listener.setImplementationType("class"); //填充属性
7 } else if (StringUtils.isNotEmpty(xtr.getAttributeValue(null, "expression"))) {
8     listener.setImplementation(xtr.getAttributeValue(null, "expression"));
9     listener.setImplementationType("expression");
10 } else if (StringUtils.isNotEmpty(xtr.getAttributeValue(null, "delegateExpression"))) {
11     listener.setImplementation(xtr.getAttributeValue(null, "delegateExpression"));
12     listener.setImplementationType("delegateExpression");
13 }
14 listener.setEvent(xtr.getAttributeValue(null, "event")); //获取 event 事件并填充对象属性
15 addListenerToParent(listener, parentElement); //将其添加到父元素
16 parseChildElements(xtr, listener, model, new FieldExtensionParser()); //解 activiti:field
17 }
18 public abstract void addListenerToParent(ActivitiListener listener, BaseElement parentElement);

```

通过上面的代码可以看出执行监听器的解析处理流程非常清晰。

(1) 第 2 行实例化 ActivitiListener 类。

listener 对象承载了执行监听器和任务监听器的存取工作,这两种类型的监听器仅是事件以及类型不同而已,其他的属性大体相同。

(2) 第 4~12 行填充对象属性。

根据监听器的创建方式执行不同的处理逻辑,关于这里一点可以参考第 11 章。

(3) 第 15 行将监听器添加到父级元素对象中。

因为监听器作为子元素存在,所以需要将 ActivitiListener 实例对象通过 addListenerToParent 方法添加到监听器的父级元素中,因为 addListenerToParent 方法是当前类的一个抽象方法,所以下面以任务监听器的实现为例详细分析该方法的处理逻辑如代码清单 4-24 所示。执行监听器 ExecutionListenerParser 类中关于该方法的实现逻辑可参考该案例自行分析。

代码清单 4-24 TaskListenerParser.java

```

1 public void addListenerToParent(ActivitiListener listener, BaseElement parentElement) {
2     if (parentElement instanceof UserTask) {
3         ((UserTask) parentElement).getTaskListeners().add(listener);
4     }

```

上面代码的处理逻辑非常简单,直接判断当前元素是否为任务节点,如果是则通过 getTaskListeners 方法获取存储当前监听器的集合,然后将 listener 元素添加到该集合中;否则不予添加操作。

(4) 解析子元素。

看到这里可能有个疑问:监听器也存在子元素吗?答案是肯定的,监听器中可以配置变量,变量可以是具体值,也可以是表达式。ActivitiListenerParser 类中的 parseChildElements 方法主要负责解析监听器中 field 元素,field 元素的配置形如 < activiti:field name = "shareniu" expression = "\${shareniu}" stringValue = "shareniu"></activiti:field>,通过

该方法中的输入参数类型可以知道 field 元素的解析工作交给 FieldExtensionParser 类完成,接下来分析该解析器的核心处理逻辑,如代码清单 4-25 所示。

代码清单 4-25 FieldExtensionParser.java

```

1 public boolean accepts(BaseElement element){
2     //该方法决定监听器是否可以添加到父元素
3     return ((element instanceof ActivitiListener) || (element instanceof ServiceTask) || (element
instanceof SendTask));
4 }
5 public void parseChildElement(XMLStreamReader xtr, BaseElement parentElement, BpmnModel
model) {
6     if (!accepts(parentElement)) return;
7     FieldExtension extension = new FieldExtension();           //activiti:field 元素的承载类
8     BpmnXMLUtil.addXMLLocation(extension, xtr);
9     extension.setFieldName(xtr.getAttributeValue(null, "name"));
10    if (StringUtils.isNotEmpty(xtr.getAttributeValue(null, "stringValue"))) {
11        extension.setStringValue(xtr.getAttributeValue(null, "stringValue"));
12    } else if (StringUtils.isNotEmpty(xtr.getAttributeValue(null, "expression"))) {
13        extension.setExpression(xtr.getAttributeValue(null, "expression"));
14    } else {
15        boolean readyWithFieldExtension = false;
16        try {
17            while (readyWithFieldExtension == false && xtr.hasNext()) {
18                xtr.next();
19                if (xtr.isStartElement() && "string".equalsIgnoreCase(xtr.getLocalName())) {
20                    extension.setStringValue(xtr.getElementText().trim());
21                } else if (xtr.isStartElement() && "expression".equalsIgnoreCase(xtr.getLocalName())) {
22                    extension.setExpression(xtr.getElementText().trim());
23                } else if (xtr.isEndElement() && getLocalName().equalsIgnoreCase(xtr.
getLocalName())) {
24                    readyWithFieldExtension = true;
25                }
26            }
27        } catch (Exception e) {
28        }
29    }
30    if (parentElement instanceof ActivitiListener) {
31        ((ActivitiListener) parentElement).getFieldExtensions().add(extension);
32    } else if (parentElement instanceof ServiceTask) {
33        ((ServiceTask) parentElement).getFieldExtensions().add(extension);
34    } else {
35        ((SendTask) parentElement).getFieldExtensions().add(extension);
36    }
37 }

```

field 元素的解析处理逻辑如下。

- (1) 第 6 行调用 accepts 方法验证 field 元素是否可以解析。
- (2) 通过 accepts 方法可以看出目前只有 ActivitiListener、ServiceTask、SendTask 可以

定义 field 元素。

(3) 第 7 行实例化 field 元素的属性承载类 FieldExtension。

(4) 第 9~13 行获取 field 元素中的 name、stringValue、expression 属性值并将其填充到 FieldExtension 实例对象中。

(5) 第 17~26 行循环遍历 field 元素中的子元素 string、expression 并将其填充到 FieldExtension 实例对象中。看到这里可能会有疑问：步骤(3)不是已经完成属性填充工作了吗？怎么这里还需要进行属性填充呢？步骤(3)解析的是 field 元素中的属性形如<activiti:field name="a" expression="" stringValue=""></activiti:field>，步骤(4)解析的是 field 元素中的子元素，形如：<activiti:field name="a"><activiti:string>a</activiti:string><activiti:expression>s</activiti:expression></activiti:field>，如果掌握了 field 元素是如何进行配置的，相信该处理逻辑也很容易理解。

(6) 第 30~36 行根据 parentElement 类型进行区分处理并将 extension 对象添加到父类中。

4.6 节点与连线关联

上文以 sequenceFlow 和 process 元素的执行监听器解析为例，讲解了流程文档中的元素解析以及属性解析，通过上文一系列讲解，可以发现元素解析之后，会将解析结果添加到父级元素（如 process 或者 subProcess）中，有这样一个问题：节点与连线何如关联？再次分析代码清单 4-11 中第 47~52 行的操作，并将其处理流程总结如下。

(1) 循环遍历所有已经解析完毕的 process 对象，如果流程文档中定义有 participant 元素（泳道），则循环遍历所有的泳道对象 pool，并判断 process 对象中的 id 值是否与 pool 对象中的 processRef 值相等，如果两者相等，则设置 pool 对象中的 executable 属性值（是否可以执行）。

(2) 调用 processFlowElements 方法进行节点与连线的关联操作，具体实现如代码清单 4-26 所示。

代码清单 4-26 BpmnXMLConverter.java

```

1 private void processFlowElements(Collection<FlowElement> flowElementList, BaseElement
2 parentScope) {
3     for (FlowElement flowElement : flowElementList) {
4         if (flowElement instanceof SequenceFlow) {
5             SequenceFlow sequenceFlow = (SequenceFlow) flowElement;
6             FlowNode sourceNode = getFlowNodeFromScope(sequenceFlow.getSourceRef(), parentScope);
7             if (sourceNode != null) {
8                 sourceNode.getOutgoingFlows().add(sequenceFlow);
9             }
10            FlowNode targetNode = getFlowNodeFromScope(sequenceFlow.getTargetRef(), parentScope);
11            if (targetNode != null) {
12                targetNode.getIncomingFlows().add(sequenceFlow);
13            }
14        } else if (flowElement instanceof BoundaryEvent) {

```

```

15 BoundaryEvent boundaryEvent = (BoundaryEvent) flowElement;
16 FlowElement attachedToElement = getFlowNodeFromScope(boundaryEvent.getAttachedToRefId()),
17 parentScope);
18 if(attachedToElement != null) {
19     boundaryEvent.setAttachedToRef((Activity) attachedToElement);
20     ((Activity) attachedToElement).getBoundaryEvents().add(boundaryEvent);
21 }
22 } else if(flowElement instanceof SubProcess) {
23     SubProcess subProcess = (SubProcess) flowElement;
24     processFlowElements(subProcess.getFlowElements(), subProcess);
25 }
26 }
27 }

```

该方法的关联节点与连线的处理流程如下。

(1) 循环遍历 flowElementList 集合。

(2) 如果 flowElement 对象类型为 SequenceFlow, 则首先获取连线中的源节点 sourceNode, 并将 sequenceFlow 对象设置到 sourceNode 对象中, 然后获取到连线中的目标节点 targetNode, 并将 sequenceFlow 对象设置到 targetNode 对象中。

(3) 如果 flowElement 对象类型为 BoundaryEvent(边界事件), 则需要将边界事件与其吸附的对象进行相互关联。

(4) 如果 flowElement 对象类型为 SubProcess(子流程), 则调用 processFlowElements 方法继续执行以上两个步骤。

```

1 private void processFlowElements(Collection<FlowElement> flowElementList, BaseElement
2 parentScope) {
3     for (FlowElement flowElement : flowElementList) {
4         if (flowElement instanceof SequenceFlow) {
5             SequenceFlow sequenceFlow = (SequenceFlow) flowElement;
6             FlowNode sourceNode = getFlowNodeFromScope(sequenceFlow.getSourceId(), parentScope);
7             if (sourceNode != null) {
8                 sourceNode.getOutgoingFlows().add(sequenceFlow);
9             }
10            FlowNode targetNode = getFlowNodeFromScope(sequenceFlow.getTargetId(), parentScope);
11            if (targetNode != null) {
12                targetNode.getIncomingFlows().add(sequenceFlow);
13            }
14            } else if (flowElement instanceof SubProcess) {
15                SubProcess subProcess = (SubProcess) flowElement;
16                processFlowElements(subProcess.getFlowElements(), subProcess);
17            }
18        }
19    }

```

第 5 章

自定义元素解析

前面提到过 Activiti 中存在默认元素和自定义元素,并详细分析了元素的解析逻辑,但是实际项目开发中,可能 Activiti 提供的元素不能完全满足开发者的需求,因此会有两个问题:一是是否可以自定义元素,二是如果元素的属性不能满足需求,是否可以扩展元素的属性。让我们带着以上两个问题开始本章的学习之旅。

5.1 自定义元素解析原理

首先讲解自定义元素的解析入口,这一过程从扩展元素 extensionElements 的解析开始,自定义元素的解析过程如代码清单 5-1 所示。

代码清单 5-1 BpmnXMLUtil.java

```
1 public static ExtensionElement parseExtensionElement(XMLStreamReader xtr) throws Exception{
2     ExtensionElement extensionElement = new ExtensionElement();//实例化扩展元素的承载类
3     extensionElement.setName(xtr.getLocalName()); //获取元素名称
4     if (StringUtils.isEmpty(xtr.getNamespaceURI())) {
5         extensionElement.setNamespace(xtr.getNamespaceURI()); //获取元素的命名空间
6     }
7     if (StringUtils.isEmpty(xtr.getPrefix())) {
8         extensionElement.setNamespacePrefix(xtr.getPrefix()); //获取元素的命名空间前缀
9     }
10    for (int i = 0; i < xtr.getAttributeCount(); i++) { //开始遍历元素的属性值
11        ExtensionAttribute extensionAttribute = new ExtensionAttribute(); //元素属性承载类
12        extensionAttribute.setName(xtr.getAttributeLocalName(i)); //属性名称
13        extensionAttribute.setValue(xtr.getAttributeValue(i)); //属性值
14        if (StringUtils.isEmpty(xtr.getAttributeNamespace(i))) {
15            extensionAttribute.setNamespace(xtr.getAttributeNamespace(i)); //属性的命名空间
16        }
17        if (StringUtils.isEmpty(xtr.getAttributePrefix(i))) {
```



```

18 //属性的命名空间前缀
19 extensionAttribute.setNamespacePrefix(xtr.getAttributePrefix(i)); }
20 // 将 extensionAttribute 设置到 extensionElement 中
21 extensionElement.addAttribute(extensionAttribute);
22 }
23 boolean readyWithExtensionElement = false;
24 while (readyWithExtensionElement == false && xtr.hasNext()) {
25     xtr.next();
26     if (xtr.isCharacters() || XMLStreamReader.CDATA == xtr.getEventType()) {
27         if (StringUtils.isNotEmpty(xtr.getText().trim())) {
28             extensionElement.setElementText(xtr.getText().trim()); //获取元素的内容
29         }
30     } else if (xtr.isStartElement()) {
31         //如果自定义元素中存在子元素,需要解析子元素并将其解析结果添加到父类中
32         ExtensionElement childExtensionElement = parseExtensionElement(xtr);
33         extensionElement.addChildElement(childExtensionElement);
34     } else if (xtr.isEndElement() &&
35         extensionElement.getName().equalsIgnoreCase(xtr.getLocalName())) {
36         readyWithExtensionElement = true;
37     }
38 }
39 return extensionElement;
40 }

```

在上述代码中,第2行实例化 ExtensionElement 类,该类作为扩展元素的属性承载类存在,第3行获取元素的名称,第4~5行获取命名空间,第7~8行获取元素的命名空间前缀,第10~21行遍历并解析该元素下的所有属性(名称、属性值、命名空间以及元素中的子元素),该操作隐含透露如下信息:自定义元素的属性个数不限,元素定义多少个属性,程序就解析多少个属性。第32行开始解析当前元素中的子元素信息。

5.2 存储自定义元素属性值

在代码清单 4-19 中, parentElement.addExtensionElement(extensionElement) 方法的主要功能是将自定义元素的解析结果存储到父级元素中,该方法的具体实现如代码清单 5-2 所示。

代码清单 5-2 BaseElement.java

```

1 protected Map<String, List<ExtensionElement>> extensionElements
2 public void addExtensionElement(ExtensionElement extensionElement) {
3     if (extensionElement != null && StringUtils.isNotEmpty(extensionElement.getName())) {
4         List<ExtensionElement> elementList = null;
5         //扩展元素集合可以为多个,所以 List 存储
6         if (this.extensionElements.containsKey(extensionElement.getName()) == false) {
7             //集合初始化
8             elementList = new ArrayList<ExtensionElement>();
9             this.extensionElements.put(extensionElement.getName(), elementList);

```

```

9      }
10     this.extensionElements.get(extensionElement.getName()).add(extensionElement);
11 }
12 }

```

addExtensionElement 方法位于 BaseElement 类中, BaseElement 类作为所有元素属性承载类的父类存在, 由此可知, 所有的流程元素都可以扩展, 例如任务节点, 任务节点的属性承载类 UserTask 就是 BaseElement 类的子类之一。上述代码中, 第 1 行代码定义的 extensionElements 集合负责存储元素的扩展信息, 所以自定义元素的信息值可以通过 BaseElement 实例对象获取。接下来讲解如何使用自定义元素。

5.3 自定义元素实战

先思考如下几个问题。

- (1) 如何自定义元素? 在哪里定义?
- (2) 自定义元素的名称以及命名空间前缀如何定义?
- (3) 自定义元素添加到父级元素之后如何进行获取?

在使用 Activiti 的时候, 开发人员有时为了满足自己的业务需求, 需要对流程文档中的 userTask 元素进行扩展, 例如期望可以在任务节点中配置该任务的处理人, 节点跳转操作等附属属性, 如果该节点可以进行跳转操作, 则需要配置可以跳转的节点集合、范围等一些信息, 这样任务节点解析时就可以将扩展的属性解析出来, 并将其注入流程虚拟机中, 从而保证流程运转时可以获取到自定义的扩展信息。下面定义一个流程文档, 该流程文档的内容如代码清单 5-3 所示。

代码清单 5-3 ShareniuExtensionElement.bpmn20.xml

```

1 <definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xmlns:activiti="http://activiti.org/bpmn"
4   xmlns:shareniu="http://www.shareniu.com"
5   typeLanguage="http://www.w3.org/2001/XMLSchema"
6   expressionLanguage="http://www.w3.org/1999/XPath" targetNamespace="shareniu">
7   <process id="shareniu" name="shareniu" isExecutable="true">
8     <startEvent id="start" name="Start"></startEvent>
9     <sequenceFlow id="flow1" sourceRef="start" targetRef="operationTask"></sequenceFlow>
10    <userTask id="operationTask" name="operationTask"
11      shareniu:id="shareniuId" shareniu:name="shareniuName">
12      <extensionElements>
13        <shareniu:shareniuOperations>
14          <shareniu:transfer transferTo="shareniu"></shareniu:transfer>
15          <shareniu:goback backTo="startevent1"></shareniu:goback>
16          <shareniu:text>shareniu</shareniu:text>
17        </shareniu:shareniuOperations>
18      </extensionElements>
19    </userTask>

```

```

20 <endEvent id = "end" name = "End" shareniu:id = "shareniuId" shareniu:name = "shareniuName">
21 </endEvent>
22 <sequenceFlow id = "flow2" sourceRef = "operationTask" targetRef = "end"></sequenceFlow>
23 </process>
24 </definitions>

```

在上述代码中,第 4 行定义了名称为 `xmlns:shareniu` 的命名空间,尽管这个命名空间看起来像一个 URL,但这并不意味着在流程文档中声明和使用命名空间的时候一定要连接到互联网,实际上命名空间仅仅是一个概念而已,可以定义为任意值,因此在浏览器中输入命名空间的 URL 并不要期望它会输出该命名空间下的所有元素和属性信息。

约定

本书中如果没有特殊说明,`xmlns:shareniu` 的值均为 `http://www.shareniu.com`。

第 12~18 行中为 id 为 `operationTask` 的任务节点添加了自定义元素,为了保证流程文档中自定义元素可以被引擎正常地解析使用,这里使用了 `shareniu` 命名空间作为自定义元素的前缀,形如 `<shareniu:shareniuOperations>`,这样操作是为了方便 Activiti 识别自定义元素,当然也可以使用 `definitions` 元素中定义的任何一个命名空间作为扩展元素的前缀,例如 `<activiti:shareniuOperations>`。

第 13 行明确告诉流程引擎 `<shareniu:shareniuOperations>` 是扩展元素(元素的名称不固定可以是任意值,千万不要以该案例定义的元素名称固化自己的思维),第 14 行定义了当前任务节点可以委托给哪一个用户进行处理,第 15 行定义了当前任务节点可以跳转到哪一个节点,第 16 行定义了一个文本标签,文本内容默认为 `shareniu`。流程文档定义完毕,接下来讲解如何获取扩展元素中的属性值,4.4 节讲解了流程文档解析完毕后所有的元素对象均存储在流程模型 `BpmnModel` 实例对象中,如果能够获取该实例对象,就可以通过该实例对象获取 id 为 `operationTask` 任务节点的所有信息,接下来自定义一个类用于获取扩展元素的属性值,该类的相关定义如代码清单 5-4 所示。

代码清单 5-4 App.java

```

1 public void readXMLFile() {
2     // 流程文档
3     String resource = "com/shareniu/chapter5/ShareniuExtensionElement.bpmn20.xml";
4     // 获取流程文档数据流
5     InputStream xmlStream = this.getClass().getClassLoader().getResourceAsStream(resource);
6     InputStreamSource xmlSource = new InputStreamSource(xmlStream);
7     // 实例化 BpmnXMLConverter
8     BpmnXMLConverter bpmnXMLConverter = new BpmnXMLConverter();
9     // 转换为流程模型
10    BpmnModel bpmnModel = bpmnXMLConverter.convertToBpmnModel(xmlSource,
11        true, true, "UTF-8");
12    // 获取 id 为 operationTask 的任务节点所有信息
13    FlowElement flowElement = bpmnModel.getProcesses().get(0)
14        .getFlowElement("operationTask");
15    // 获取扩展元素的信息

```

```
16 Map<String, List<ExtensionElement>> extensionElements = flowElement
17 .getExtensionElements();
18 Iterator<Entry<String, List<ExtensionElement>>> it = extensionElements
19 .entrySet().iterator();
20 while (it.hasNext()) {
21     Entry<String, List<ExtensionElement>> entry = it.next();
22     // 获取根元素名称
23     System.err.println("rookey = " + entry.getKey());
24     List<ExtensionElement> value = entry.getValue();
25     for (ExtensionElement e : value) {
26         Map<String, List<ExtensionElement>> childElements = e.getChildElements();
27         Iterator<Entry<String, List<ExtensionElement>>> it1 = childElements
28             .entrySet().iterator();
29         while (it1.hasNext()) {
30             Entry<String, List<ExtensionElement>> entry1 = it1.next();
31             System.err.println("childKey = " + entry1.getKey());
32             List<ExtensionElement> value1 = entry1.getValue();
33             for (ExtensionElement el : value1) {
34                 String elementText = el.getElementText(); // 获取文本
35                 System.err.println(elementText);
36                 Map<String, List<ExtensionAttribute>> attributes = el.getAttributes();
37                 Collection<List<ExtensionAttribute>> values = attributes.values();
38                 for (List<ExtensionAttribute> list : values) {
39                     for (ExtensionAttribute extensionAttribute : list) {
40                         System.err.println(extensionAttribute.getName() + ", " + extensionAttribute.getValue());
41                     }
42                 }
43             }
44         }
45     }
46 }
```

在上述代码中,第3~6行读取代码清单5-3定义的流程文档数据流,第10~11行开始获取BpmnModel实例对象,第13~14行获取id为operationTask任务节点的所有信息,第16~17行获取该任务节点的所有自定义元素。

执行上述代码后,控制台打印的信息如下:

```
rookey = shareniuOperations
childKey = transfer
transferTo, shareniu
childKey = goback
backTo, startevent1
childKey = text
```

5.4 扩展黑名单元素

在4.4.3节介绍process、userTask、definitions元素解析时使用了黑名单处理机制, userTask节点的解析工作是围绕其对应的解析器UserTaskXMLConverter中的convertXMLToElement方法展开的,该方法首先解析任务节点中定义的常规属性,然后委托BpmnXMLUtil类中的静态方法addCustomAttributes解析自定义属性。这里以扩展任务节点属性为例,详细讲解该实现过程。

5.4.1 扩展元素属性原理

BpmnXMLUtil 类中的 addCustomAttributes 方法相关实现如代码清单 5-5 所示。

代码清单 5-5 BpmnXMLUtil.java

```
1 public static void addCustomAttributes (XMLStreamReader xtr, BaseElement element, List
<ExtensionAttribute>... blackLists) {
2     for (int i = 0; i < xtr.getAttributeCount(); i++) {
3         ExtensionAttribute extensionAttribute = new ExtensionAttribute(); //扩展属性
4         extensionAttribute.setName(xtr.getAttributeLocalName(i)); //属性的名称
5         extensionAttribute.setValue(xtr.getAttributeValue(i)); //属性值
6         if (StringUtils.isEmpty(xtr.getAttributeNamespace(i))) { //命名空间
7             extensionAttribute.setNamespace(xtr.getAttributeNamespace(i));
8         }
9         if (StringUtils.isEmpty(xtr.getAttributePrefix(i))) { //命名空间前缀
10            extensionAttribute.setNamespacePrefix(xtr.getAttributePrefix(i));
11        }
12        //element 为 userTask
13        if (!isBlacklisted(extensionAttribute, blackLists)) { //黑名单校验
14            element.addAttribute(extensionAttribute);
15        }
16    }
17 }
```

在上述代码中，第 2~15 行循环遍历任务节点的所有属性信息，其中第 13 行验证黑名单列表，如果属性不在黑名单列表中，则执行第 14 行代码，将其作为扩展属性信息进行存储。任务节点中定义的黑名单集合如表 5-1 所示。

表 5-1 任务节点的黑名单集合

属 性	含 义
id	id 值，必须全局唯一
name	任务节点的名称
activiti:async	是否异步执行任务节点
isForCompensation	是否补偿
activiti:formKey	为任务节点绑定 form
activiti:dueDate	任务到期时间
activiti:candidateUsers	候选人
activiti:priority	属性
activiti:category	分类
activiti:skipExpression	跳过表达式

5.4.2 任务节点扩展属性实战

下面定义一个流程文档以验证自定义黑名单是否生效，该流程文档内容如代码清单 5-6

所示。

代码清单 5-6 customTask.bpmn

```
1 <process id="myProcess" name="shareniu" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="operationTask" name="operationTask" shareniu:id="shareniuId" shareniu:
name="shareniuName"></userTask>
4   <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="operationTask">
</sequenceFlow>
5   <endEvent id="endevent1" name="End"></endEvent>
6   <sequenceFlow id="flow2" sourceRef="operationTask" targetRef="endevent1">
</sequenceFlow>
7 </process>
```

在上述代码中,第3行为任务节点定义了两个扩展属性 shareniu:id 和 shareniu:name,其中 shareniu 为代码清单 5-3 中定义的命名空间,任务节点的扩展属性需要引入命名空间前缀以方便与任务节点的默认属性进行区分,扩展属性的名称不限于上面两种,可以是任意值。下面自定义一个类用于获取任务节点的扩展属性,如代码清单 5-7 所示。

代码清单 5-7 App.java

```
1 public void getAttributes() {
2   String resource = "com/shareniu/chapter5/customTask.bpmn";
3   //文件流
4   InputStream xmlStream = this.getClass().getClassLoader()
5     .getResourceAsStream(resource);
6   InputStreamSource xmlSource = new InputStreamSource(xmlStream);
7   // 实例化 BpmnXMLConverter
8   BpmnXMLConverter bpmnXMLConverter = new BpmnXMLConverter();
9   // 转换为流程模型
10  BpmnModel bpmnModel = bpmnXMLConverter.convertToBpmnModel(xmlSource,
11    true, false, "UTF-8");
12  //获取 id 为 operationTask 任务节点的所有信息
13  FlowElement flowElement = bpmnModel.getProcesses().get(0)
14    .getFlowElement("operationTask");
15  Map<String, List<ExtensionAttribute>> attributes = flowElement.getAttributes();
16  List<ExtensionAttribute> list = attributes.get("id");
17  String name = list.get(0).getName();
18  String value = list.get(0).getValue();
19  System.out.println(name + "," + value);
20  list = attributes.get("name");
21  name = list.get(0).getName();
22  value = list.get(0).getValue();
23  System.out.println(name + "," + value);
24 }
```

执行上述代码,控制台打印的信息如下:

```
id,shareniuId
name,shareniuName
```

强调

shareniu:id 与 id 两个属性的命名空间前缀不一致,因此引擎将 shareniu:id 作为扩展属性处理。

5.5 扩展非黑名单元素

上文提到过只有 process、userTask、definitions 元素解析的时候,Activiti 才会使用黑名单机制对自定义属性进行解析和存储,看到这里可能会有疑问,除了上面所说的三种元素,Activiti 是否也对其他元素的扩展属性进行了处理?有点遗憾,Activiti 并没有对其他元素的扩展属性进行处理,如果开发者期望在其他元素中使用自定义属性,该如何操作?

5.5.1 自定义元素解析器

这里以开始节点(StartEvent 元素)的自定义属性解析为例,详细讲解开始节点的自定义属性解析过程,为了简单起见,直接自定义开始节点解析器,并继承 StartEventXMLConverter (开始节点的默认解析器),相关实现如代码清单 5-8 所示。

代码清单 5-8 ShareniuStartEventXMLConverter.java

```
1 public class ShareniuStartEventXMLConverter extends StartEventXMLConverter {
2     protected static final List<ExtensionAttribute> defaultElementAttributes = Arrays.
asList(new
3         ExtensionAttribute(ATTRIBUTE_ID), new ExtensionAttribute(ATTRIBUTE_NAME));
4     public Class<? extends BaseElement> getBpmnElementType() {
5         return StartEvent.class;
6     }
7     protected String getXMLElementName() {
8         return "startEvent";
9     }
10    BaseElement convertXMLToElement(XMLStreamReader xtr, BpmnModel model) {
11        String formKey = xtr.getAttributeValue(ACTIVITI_EXTENSIONS_NAMESPACE, "formKey");
12        StartEvent startEvent = null;
13        if (StringUtils.isEmpty(formKey)) {
14            if (model.getStartEventFormTypes() != null
15                && model.getStartEventFormTypes().contains(formKey)) {
16                startEvent = new AlfrescoStartEvent();
17            }
18        }
19        if (startEvent == null) {
20            startEvent = new StartEvent();
21        }
22        BpmnXMLUtil.addXMLLocation(startEvent, xtr);
23        startEvent.setInitiator(xtr.getAttributeValue(ACTIVITI_EXTENSIONS_NAMESPACE,
24            "initiator"));
25        startEvent.setFormKey(formKey);
26        BpmnXMLUtil.addCustomAttributes(xtr, startEvent, defaultElementAttributes);
27    }
28 }
```

```

27 parseChildElements(getXMLElementName(), startEvent, model, xtr);
28 return startEvent;
29 }
30 }

```

在上述代码中,第2~3行初始化了黑名单集合 defaultElementAttributes,该集合定义了 id、name 两个属性,这样引擎解析开始节点时,除了以上定义的两个属性之外,其他属性都会作为扩展属性进行处理,第4~6行告诉引擎开始节点属性信息使用 StartEvent 类进行封装,第7~9行告诉引擎程序要解析 startEvent 类型的元素,第10行中定义的 convertXMLToElement 方法负责解析开始节点,该过程比较复杂,其处理逻辑如下。

(1) 解析常规属性。

第11行解析常规属性,例如 formKey。第13~21行根据 formKey 属性值的有无,实例化不同的类,第22行解析并存储开始节点在流程文档中的坐标信息,第23~24行设置 initiator 属性值,initiator 属性的配置形如 activiti:initiator="shareniu",这样启动流程实例时,该属性会作为流程实例级别的变量存在。该属性的处理可以参考 13.3 节。

(2) 解析自定义属性。

第26行调用 BpmnXMLUtil 类中的静态方法 addCustomAttributes 进行自定义属性的解析工作。

(3) 解析子元素。

第27行调用 parseChildElements 方法进行子元素的解析工作。

可能阅读完上面的代码会有疑惑,上述自定义开始节点的解析器就是直接将 StartEventXMLConverter 类中的解析代码复制过来然后添加了第26行代码,为何不直接调用父类的方法,如代码清单 5-9 所示。

代码清单 5-9 ShareniuStartEventXMLConverter.java

```

1 protected BaseElement convertXMLToElement(XMLStreamReader xtr, BpmnModel model) {
2     BaseElement element = super.convertXMLToElement(xtr, model);
3     BpmnXMLUtil.addCustomAttributes(xtr, element, defaultElementAttributes);
4     return element;
5 }

```

在上述代码中,convertXMLToElement 方法的处理逻辑为:第2行委托父类的 convertXMLToElement 方法进行元素解析工作并返回解析结果 element;第3行实现自定义属性的解析。

上面的代码处理逻辑看起来很合理,也很清晰,但是这样操作是完全错误的,该案例中 convertXMLToElement 方法主要用于解析开始节点,第4章中讲解了流模型解析文档的原理,使用流模型解析文档时读取到的数据流只能前进不能回退,如果第2行委托父类进行元素的解析工作则势必会涉及子元素的解析,如果执行完子元素的解析工作之后,再次执行第3行解析父类的属性,这样的操作是完全错误,因为流模型解析文档的时候数据流只能前进不能回退。

该案例也从侧面说明一个问题,了解原理方能看透本质,才能更好地运用框架提供的

功能。

5.5.2 替换引擎元素解析器

接下来使用自定义开始节点解析器以替换开始节点的默认解析器 StartEventXMLConverter, 相关实现如代码清单 5-10 所示。

代码清单 5-10 App.java 和 startEvent.bpmn

```

1 com/shareniu/chapter5/startEvent.bpmn:
2 <process id="myProcess" name="My process" isExecutable="true">
3   <startEvent id="startevent1" shareniu:id="shareniuId" shareniu:name="shareniuName">
4   </startEvent>
5   <userTask id="operationTask" name="operationTask"></userTask>
6   <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="operationTask">
7   </sequenceFlow>
8   <endEvent id="endevent1" name="End"></endEvent>
9   <sequenceFlow id="flow2" sourceRef="operationTask" targetRef="endevent1">
10  </sequenceFlow>
11 </process>
12 App.java:
13 public void sequencegetAttributes() {
14   // 流程文档
15   String resource = "com/shareniu/chapter5/startEvent.bpmn";
16   // 文件流
17   InputStream xmlStream = this.getClass().getClassLoader()
18     .getResourceAsStream(resource);
19   InputStreamSource xmlSource = new InputStreamSource(xmlStream);
20   // 实例化 BpmnXMLConverter
21   BpmnXMLConverter bpmnXMLConverter = new BpmnXMLConverter();
22   BpmnXMLConverter.addConverter(new ShareniuStartEventXMLConverter());
23   // 转换为流程模型
24   BpmnModel bpmnModel = bpmnXMLConverter.convertToBpmnModel(xmlSource,
25     true, false, "UTF-8");
26   // 获取 id 为 operationTask 任务节点的所有信息
27   FlowElement flowElement = bpmnModel.getProcesses().get(0)
28     .getFlowElement("startevent1");
29   Map<String, List<ExtensionAttribute>> attributes = flowElement.getAttributes();
30   List<ExtensionAttribute> list = attributes.get("id");
31   String name = list.get(0).getName();
32   String value = list.get(0).getValue();
33   System.out.println(name + "," + value);
34   list = attributes.get("name");
35   name = list.get(0).getName();
36   value = list.get(0).getValue();
37   System.out.println(name + "," + value);
38 }

```

在上述代码中,第 2~11 行定义了流程文档 startEvent.bpmn,其中第 3 行为开始节点

定义了 `shareniu:id`、`shareniu:name` 两个扩展属性。第 22 行使用自定义开始节点解析器，替换引擎默认实现，该步骤非常重要。执行上述代码，控制台打印的信息如下：

```
id, shareniuId
name, shareniuName
```

本节主要讲解了自定义元素、自定义任务节点的属性以及自定义元素解析器的处理过程，到此为止 Activiti 中所有的元素以及元素的解析工作已经讲解完毕。

扩展

其他元素扩展属性的实现过程可以结合该案例进行深入学习。

的类或者具体值进行初始化工作。事件转发器的处理也不例外，引擎默认使用的事件转发器为 `ActivitiEventDispatcherImpl` 类，当然也可以通过设置 `ProcessEngineConfigurationImpl` 类中的 `eventDispatcher` 属性值注入自定义事件转发器。

(2) 开启事件转发功能

第 9 行设置 `eventDispatcher` 对象的属性 `enabled` 值为 `true`，该属性的默认值为 `false`，即引擎默认关闭了事件转发功能。如果其值为 `true`，则引擎会启动 `Activiti` 中的事件转发功能，并针对不同的事件进行不同的事件转发器选择。通常来说，在流程图中，当遇到一个任务时，引擎会根据该任务的类型，选择相应的事件转发器。本章主要讲解的是如何自定义事件转发器，并使其生效。

(3) 注册全局事件监听器

首先第 10 行判断 `eventListeners` 集合是否为空，如果该集合不为空，则说明已经注册了全局事件监听器。此时，我们不需要再注册新的监听器。如果该集合为空，则需要注册新的全局事件监听器。

(4) 注册具体类型的事件监听器

步骤(3)负责注册全局事件监听器，而步骤(4)负责注册具体类型的事件监听器。在 `ActivitiEventDispatcherImpl` 类中，`addEventListeners` 方法负责为 `eventListeners` 集合添加具体的事件监听器。该方法的实现逻辑如下：首先，根据传入的事件类型，从 `eventListeners` 集合中查找对应的监听器。如果找到，则将其添加到 `eventListeners` 集合中。如果没有找到，则需要创建一个新的监听器，并将其添加到 `eventListeners` 集合中。最后，返回 `eventListeners` 集合。

```
10 if (this.eventListeners == null) {
11     for (ActivitiEventListener listenerToAdd : eventListeners) {
12         this.eventDispatcher.addEventListeners(listenerToAdd);
13     }
14 }
```

在 `initEventDispatcher` 方法中的第二个参数 `eventListeners` 是一个 `ActivitiEventListener` 类型的集合。该集合用于存储所有的事件监听器。在 `addEventListeners` 方法中，我们遍历这个集合，并将每个监听器添加到 `eventListeners` 集合中。最后，返回 `eventListeners` 集合。

第 6 章

接下来使用自定义开始节点解析器以替换开始节点的默认解析器 StartEvent。本章以事件转发器的初始化过程作为切入点，深入分析其内部实现机制。

事件转发器

```

1 <startEvent id="startEvent" sharedId="startEvent" name="startEvent">
2   </startEvent>
3   <task id="operationTask" name="operationTask">
4     <sequenceFlow id="flow1" sourceRef="startEvent" targetRef="operationTask">
5     </sequenceFlow>
6   </task>
7   <endEvent id="endEvent1" name="End">
8     <sequenceFlow id="flow2" sourceRef="operationTask" targetRef="endEvent1">
9     </sequenceFlow>

```

在前面的讲解中，经常看到流程引擎使用事件转发器进行不同事件的转发工作，那么事件转发器是如何工作的呢？它的工作原理是什么呢？本章以事件转发器的初始化过程作为切入点，深入分析其内部实现机制。

6.1 初始化事件转发器

事件转发器的初始化过程如代码清单 6-1 所示。

代码清单 6-1 ProcessEngineConfigurationImpl.java

```

1 protected boolean enableEventDispatcher = true;
2 protected ActivitiEventDispatcher eventDispatcher;
3 protected List<ActivitiEventListener> eventListeners;
4 protected Map<String, List<ActivitiEventListener>> typedEventListeners;
5 protected void initEventDispatcher() {
6   if(this.eventDispatcher == null) {
7     this.eventDispatcher = new ActivitiEventDispatcherImpl();
8   }
9   this.eventDispatcher.setEnabled(enableEventDispatcher);
10  if(eventListeners != null) {
11    for(ActivitiEventListener listenerToAdd : eventListeners) {
12      this.eventDispatcher.addEventListener(listenerToAdd);
13    }
14    if(typedEventListeners != null) {
15      for (Entry<String, List<ActivitiEventListener>> listenersToAdd : typedEventListeners.
16        entrySet()) {
17        ActivitiEventType[] types = ActivitiEventType.getTypesFromString(listenersToAdd.getKey());
18        for(ActivitiEventListener listenerToAdd : listenersToAdd.getValue()) {
19          this.eventDispatcher.addEventListener(listenerToAdd, types);

```

```

19     }
20 }
21 }
22 }

```

如果想要使用事件转发器发布事件,那么初始化事件转发器是一个必不可少的环节,下面概括总结 `initEventDispatcher` 方法的初始化逻辑。

(1) 第 6~7 行初始化前的准备工作。

在 2.4 节中详细讲了开关属性的初始化过程,简单概括为:首先判断指定的开关属性值是否为空,如果不为空则直接使用客户端指定的类或者值进行初始化,否则使用系统内置的类或者具体值进行初始化工作。事件转发器的处理也不例外,引擎默认使用的事件转发器为 `ActivitiEventDispatcherImpl` 类,当然也可以通过设置 `ProcessEngineConfigurationImpl` 类中的 `eventDispatcher` 属性值注入自定义事件转发器。

(2) 开启事件转发功能。

第 9 行设置 `eventDispatcher` 对象的属性 `enabled` 值为 `true`,从该属性的默认值可以看出,流程引擎默认开启了事件转发功能,如果开发人员不打算使用 `Activiti` 中的事件转发功能,可以将该 `enableEventDispatcher` 开关属性值设置为 `false` 从而对事件转发功能进行全局禁用。

(3) 注册全局事件监听器。

首先第 10 行判断 `eventListeners` 集合是否为空,如果该集合不为空,则第 11~12 行循环遍历该集合并调用 `eventDispatcher` 对象中的 `addEventListener` 方法注册全局事件监听器。

(4) 注册具体类型的事件监听器。

步骤(3)负责注册全局事件监听器,第 14~19 行负责注册具体类型的事件监听器,`typedEventListeners` 集合为 `Map` 数据结构,key 对应具体的事件类型,value 对应具体类型的事件监听器,处理流程比较简单,首先遍历 `typedEventListeners` 集合,第 16 行提取 key 值并将其转化为 `ActivitiEventType` 类型的数组(具体处理过程是将 key 值使用“,”进行分割,可以参考 `getTypesFromString` 方法的具体实现),最后第 17~18 行根据 `listenerToAdd` 对象和 `types` 进行具体类型的事件监听器的注册工作。

建议

在实际项目开发中,建议开发人员开启事件转发功能。

6.2 事件转发器架构

在细化讲解 `initEventDispatcher` 方法中的每一个具体细节实现之前,首先看一下 `ActivitiEventDispatcher` 接口的架构设计,如图 6-1 所示。

根据图 6-1 可以很清晰地从全局角度了解 `ActivitiEventDispatcher` 的脉络,下面简单概括每个类的作用。

(1) `ActivitiEventDispatcher`: 该接口定义了注册事件监听器、移除事件监听器、转发事

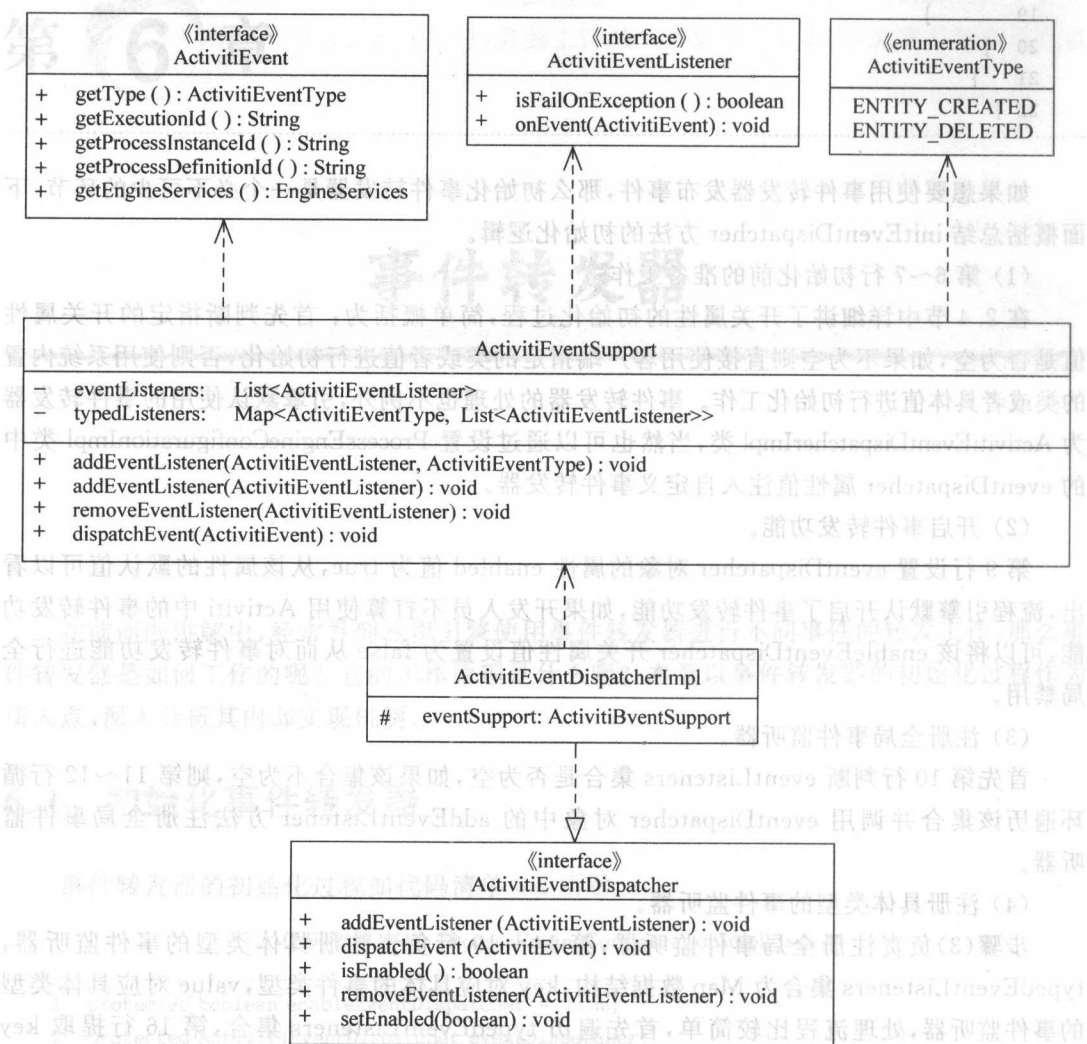


图 6-1 ActivitiEventDispatcher 类架构图

件以及设置是否开启事件转发功能等方法。

(2) **ActivitiEventSupport**: 该类内部维护一个全局事件监听器集合 `eventListeners` 和具体类型的事件监听器 `typedListeners` 集合, 并且定义了这两个集合的添加、删除方法以及转发不同事件的方法。

(3) **ActivitiEventDispatcherImpl**: 该类对 **ActivitiEventDispatcher** 接口进行实现, 其内部持有 **ActivitiEventSupport** 实例, 并可以根据当前类中的 `enabled` 属性值决定是否可以对事件进行转发。该类委托 **ActivitiEventSupport** 实例对象完成事件监听器的注册、移除和事件转发功能。由于 **ProcessEngineConfigurationImpl** 类中持有 **ActivitiEventDispatcher** 实例对象的引用, 因此开发人员可以通过 **ProcessEngineConfigurationImpl** 获取 **ActivitiEventDispatcher** 实例对象进行事件监听器的注册、移除以及事件转发工作。

(4) **ActivitiEvent**: 定义了获取事件类型、执行实例 id, 流程实例 id, 流程定义 id 的方法以及获取 **EngineServices** 实例对象的方法。所有的事件转发器均需要直接或者间接的实现

该接口。

(5) `ActivitiEventType`: 该枚举类定义了所有可以操控的事件类型。

(6) `ActivitiEventListener`: 该接口定义了转发事件的方法 `onEvent` 以及事件监听器执行时如果程序出现异常该如何处理的方法 `isFailOnException`。

注意

`ActivitiEventDispatcher` 类为默认的事件转发器, 该类负责管理所有的事件监听器。

6.3 注册事件监听器

下面详细分析全局事件监听器和具体类型的事件监听器是如何注册使用的。事件监听器从应用范围上可以划分为两种: 一种是全局事件监听器, 负责监听所有的事件; 另一种是具体类型的事件监听器, 只负责部分事件的监听, 例如开发人员可以将所有流程引擎支持的事件按照业务使用场景进行划分, 可以定义一个类只负责监听所有与任务有关的事件, 也可以定义一个类只负责变量事件的监听等, 引擎设计具体类型的事件监听器的主要意图是方便开发人员细化、归类事件, 从而使事件监听器的职责更加清晰和单一, 代码也更加容易维护。

上面提到事件监听器的注册和移除工作最终是通过 `ActivitiEventDispatcherImpl` 类中持有的 `ActivitiEventSupport` 实例对象来完成, 下面详细分析 `ActivitiEventSupport` 类是如何进行事件监听器的注册工作, 整个处理流程如代码清单 6-2 所示。

代码清单 6-2 `ActivitiEventSupport.java`

```
1 synchronized void addEventListener(ActivitiEventListener listenerToAdd) {
2     if (listenerToAdd == null) {}
3     if (!eventListeners.contains(listenerToAdd)) {
4         eventListeners.add(listenerToAdd);
5     }
6 }
7 synchronized void addEventListener(ActivitiEventListener listenerToAdd, ActivitiEventType...
types) {
8     if (listenerToAdd == null) {}
9     if (types == null || types.length == 0) {
10        addEventListener(listenerToAdd); //注册全局事件监听器
11    } else {
12        for (ActivitiEventType type : types) {
13            addTypedEventListener(listenerToAdd, type); //注册具体类型的事件监听器
14        }
15    }
16 }
17 synchronized void addTypedEventListener(ActivitiEventListener listener, ActivitiEventType
type){
18     List<ActivitiEventListener> listeners = typedListeners.get(type);
19     if (listeners == null) {
```

```

20     listeners = new CopyOnWriteArrayList<ActivitiEventListener>();
21     typedListeners.put(type, listeners);
22 }
23 if (!listeners.contains(listener)) {
24     listeners.add(listener);
25 }
26 }

```

(1) 第 1 行定义的 `addEventListener` 方法用于注册全局事件监听器。

`addEventListener` 方法主要用于注册全局事件监听器,该方法的处理逻辑非常简单,首先第 2 行对 `listenerToAdd` 参数进行非空校验,如果该参数不为空,则第 3 行判断 `eventListeners` 集合中是否已经存在了 `listenerToAdd`,如果集合中不存在,则第 4 行将其添加到集合中,该操作主要是为了避免重复注册事件监听器。

(2) 第 7 行定义的 `addEventListener` 方法用于注册具体类型的事件监听器。首先第 8 行对 `listenerToAdd` 参数进行非空校验,然后第 9 行判断 `type` 参数是否为空,如果 `type` 参数为空则第 10 行直接将当前的事件监听器作为全局事件监听器进行处理,并通过第 1 行定义的方法将其注册到 `eventListeners` 集合中;如果 `type` 参数不为空,则第 12~13 行循环遍历 `type` 数组,并调用第 17 行定义的 `addTypedEventListener` 方法注册具体类型的事件监听器,该方法的处理逻辑如下。

- 第 18 行根据 `type` 参数从 `typedListeners` 集合中获取该事件下所有的事件监听器集合 `listeners`。
- 第 19 行如果 `listeners` 集合为空,则首先初始化该集合并将其添加到 `typedListeners` 集合中。
- 第 23 行如果 `listeners` 集合中不存在 `listener` 元素,则将其添加到 `listeners` 集合中。

因为向 `eventListeners` 集合中添加元素时,可能存在多个线程同时对集合内容进行操作,所以需要考虑集合数据同步问题,因此上面三个方法中的方法签名均使用了 `synchronized` 关键字进行修饰。

`ActivitiEventSupport` 类中的 `eventListeners` 集合为 `List` 数据结构,初始化该集合时使用了 Java 中的 `CopyOnWriteArrayList` 类,该集合的初始化过程可以参考 `ActivitiEventSupport` 类的构造方法。这里需要对 `CopyOnWriteArrayList` 类详细说明: `CopyOnWriteArrayList` 类非常重要,可以使用在高并发场景中,基本思路是这样的,例如存在一个集合,多个线程都要对集合内容进行操作,当任意一个线程想要改变集合内容时,首先需要把集合中的内容进行一次全复制并形成一个新的容器,这样任何一个线程想要对原容器中的内容进行修改操作时,只会修改新生成容器中的内容,修改完毕之后将原容器的指针指向新容器,从而达到修改集合内容的目的,这是一种典型的懒情延时策略,这样做的好处就是每次修改容器内容时操作的永远是新容器,从而可以对 `CopyOnWriteArrayList` 进行并发读而不需要加同步锁,由于当前容器集合永远不会添加新元素,读写操作永远在不同的容器中进行,是典型的读写分离思想。

`CopyOnWriteArrayList` 类的缺点:在写操作过程中内存会同时存在两个容器,这样程序就会占用大量的内存,并且写操作过程客户端可能需要从原容器中读取最新的数据,而原容器的指针在指向新容器之前程序读取的永远是原容器的值,这样可能会造成数据短时间

内不一致,如果开发人员期望写数据立刻就能被读取到,必须要求保证数据的实时性、一致性,那么很显然使用 CopyOnWriteArrayList 容器不是一种明智的选择。

注意

Java 中的 Map 为引用类型,如果获取到某个 Map 的引用,则程序操作引用对象的时候,本质上就是对 Map 进行操作。

6.4 事件转发功能之新老版本兼容

上面详细讲解了事件监听器的注册过程,注册之后的事件监听器是如何被级联触发的呢?下面详细分析 ActivitiEventDispatcherImpl 类中的 dispatchEvent 方法,该方法的具体实现如代码清单 6-3 所示。

代码清单 6-3 ActivitiEventDispatcherImpl.java

```
1 public void dispatchEvent( ActivitiEvent event ){
2     if ( enabled ){
3         eventSupport.dispatchEvent( event );
4     }
5     if ( Context.isExecutionContextActive() ){
6         ProcessDefinitionEntity definition = Context.getExecutionContext().getProcess-
7             Definition();
8         if ( definition != null ){
9             definition.getEventSupport().dispatchEvent( event );
10        }
11    } else {
12        CommandContext commandContext = Context.getCommandContext();
13        if ( commandContext != null ){
14            ProcessDefinitionEntity processDefinition = extractProcessDefinitionEntity-
15                FromEvent( event );
16            if ( processDefinition != null ){
17                processDefinition.getEventSupport().dispatchEvent( event );
18            }
19        }
20    }
```

方法。该方法只是做了一些辅助性工作,核心的处理工作委托给 eventSupport 对象中的 dispatchEvent 方法,当前类中的 dispatchEvent 方法的处理逻辑如下所示。

(1) 第 2 行校验 enabled 值。

如果该值为 true,则第 3 行委托 eventSupport 对象中的 dispatchEvent 方法进行事转发,否则执行如下的逻辑。

(2) 第 5 行探测执行上下文是否存活。

第 5 行判断 ExecutionContext 实例对象是否存在,如果存在则说明流程实例正在进行运转并且流程引擎实例对象已经被初始化,第 8 行就可以通过 ExecutionContext 实例对象获取 definition 对象,关于 Context 类的更多讲解可以参考第 12 章。第 6 行判断 definition

对象是否为空,如果不为空,则第 8 行委托 definition 对象中的 eventSupport 对象进行事件的转发操作;如果 ExecutionContext 实例对象不存在,则说明流程实例还没有开始运转,那么第 11 行开始查找 commandContext 对象,如果 commandContext 对象不为空,则第 13 行获取 processDefinition 对象,如果该对象不为空,则第 15 行开始进行事件转发。引擎为何要使用 ProcessDefinitionEntity 类中的 eventSupport 对象进行事件的转发呢? ProcessDefinitionEntity 类中的 eventSupport 对象与上面讲解的 ActivitiEventDispatcherImpl 类中持有的 eventSupport 对象,是两个完全不同的对象,这是 Activiti5.15 版本之前的历史遗留问题,在 Activiti5.15 之前的版本中,事件类型以及事件监听器仅可以配置在流程文档中且只能配置在 process 元素中,如果开发人员需要修改事件监听器,那就必须对流程文档进行重新配置和部署,难度以及可维护度可想而知。关于流程文档中配置事件监听器的实现,如代码清单 6-4 所示。

代码清单 6-4 ActivitiEventListener.bpmn

```

1 <process id="myProcess" name="My process" isExecutable="true">
2   <extensionElements>
3     <!-- 全局事件监听器 -->
4     <activiti:eventListener class="com.shareniu.chapter6.ShareniuTaskEventListener"/>
5     <activiti:eventListener class="com.shareniu.chapter6.ShareniuTaskEventListener"
6       eventType="ENTITY_CREATED" />
7   </extensionElements>
8   ...//省略其他元素
9 </process>

```

在上述代码中,第 4 行定义了一个全局的事件监听器,第 5~6 行定义了一个具体类型的事件监听器。关于 process 元素中事件监听器的解析过程可以参考 4.5.3 节的讲解,流程定义实体的获取方法 extractProcessDefinitionEntityFromEvent 的处理过程可以参考 13.2.2 节。

6.5 事件转发原理以及缺陷

下面开始讲解事件转发的处理逻辑,代码清单 6-3 中事件转发工作最终会委托给 ActivitiEventSupport 类中的事件发布方法 dispatchEvent 进行处理,相关实现如代码清单 6-5 所示。

代码清单 6-5 ActivitiEventSupport.java

```

1 public void dispatchEvent(ActivitiEvent event) {
2   if (event == null || event.getType() == null) {
3     throw new ActivitiIllegalArgumentException("Event cannot be null.");
4   }
5   if (!eventListeners.isEmpty()) {
6     for (ActivitiEventListener listener : eventListeners) {
7       dispatchEvent(event, listener);
8     }
9   }
10  List<ActivitiEventListener> typed = typedListeners.get(event.getType());

```

```

9     if (typed != null && !typed.isEmpty()) {
10    for (ActivitiEventListener listener : typed) {
11        dispatchEvent(event, listener);
12    }}
13 }
14 protected void dispatchEvent(ActivitiEvent event, ActivitiEventListener listener) {
15     try {
16         listener.onEvent(event);
17     } catch (Throwable t) {
18         if (listener.isFailOnException()) {
19             throw new ActivitiException("Exception while executing event - listener", t);
20         } else { ...//省略日志输出 }
21 }

```

上面方法的主要工作就是负责事件转发,主要处理步骤总结如下。

(1) 对转发的事件进行校验,第 2 行对当前需要转发的事件以及事件类型进行非空校验,如果任意一个为空,则程序报错。试想一下转发一个空事件有何意义呢,如果事件类型为空,事件监听器又该如何检测到该类型的事件呢。其实这个地方的处理有一点小瑕疵,因为不论 event 参数为空或是 event.getType() 方法的返回值为空,第 3 行均抛出相同的异常信息,所以可能会对开发人员造成困惑,排查该异常信息时不能快速定位问题,不知道到底是因为 event 参数为空还是 event.getType() 方法的返回值为空而导致程序出错,所幸 Activiti 5.22 版本解决了这个 Bug。

(2) 按照先后顺序转发事件。

第 4 行如果 eventListeners 集合不为空则第 5 行循环遍历该集合并在第 6 行调用 dispatchEvent 方法转发事件,执行完所有的全局事件监听器之后再开始执行具体类型的事件监听器,流程如下:第 8 行根据事件类型从 typedListeners 集合中查找该事件对应的所有事件监听器集合 typed,第 9 行如果 typed 集合不为空,则循环遍历该事件类型对应的所有事件监听器并在第 11 行调用 dispatchEvent 方法进行事件转发。通过分析事件转发的处理逻辑可以得出一个结论:如果开发人员定义一个全局事件监听器 A 和具体类型的事件监听器 B 分别对事件类型为 TASK_CREATED 的事件进行监听处理,则首先执行全局事件监听器 A 然后执行具体类型的事件监听器 B,同一个事件可以对应多个事件监听器。

(3) 第 14 行定义的 dispatchEvent 方法用于转发事件。

该方法的执行逻辑非常简单但至关重要,首先第 16 行触发 listener 对象中的 onEvent 方法,并为该方法传入 event 参数值(事件所对应的类),如果程序出现异常,则第 17 行根据事件监听器中 isFailOnException 方法的返回值进行处理,如果该方法返回 true,表示程序对异常零容忍,这时程序直接报错,流程实例不会继续向下运转;如果该方法返回 false,则忽略异常信息,流程实例继续运转不受干扰。

经过上面的分析发现 Activiti 对于事件转发以及事件监听器的处理机制完全与观察者模式相吻合,因为观察者模式的本质就是“级联触发”,观察者模式说明:定义对象之间一对一或者一对多的依赖关系,当一个对象的状态发生变化时,所有依赖于它的对象都会得到通知并自动更新,上面代码中 ActivitiEventSupport 实例对象就是目标对象,该对象中定义了观察者对象 ActivitiEventListener 的添加和删除操作以及通知观察者的方法 dispatchEvent。

6.6 添加事件监听器

Activiti 中定义了一系列的事件以及事件处理类,那么事件类型有哪些?事件处理类又是如何被创建的?事件处理类的创建非常简单,由于篇幅有限,本书不过多讲解,事件类型均定义在 `ActivitiEventType` 类中,如果期望快速学习每一个事件的具体含义以及该事件的处理类,最简单的方法就是监听所有的事件并进行相应输出,进而观察每一个事件以及该事件处理类的行为。

6.6.1 使用配置方式添加

下面重点讲解自定义事件监听器的处理逻辑,首先定义一个全局事件监听器以及具体类型的事件监听器(两者使用的是同一个类)并将该类注入流程引擎配置类,如代码清单 6-6 所示。

代码清单 6-6 ShareniuEventListener.java 和 activiti.cfg.xml

1 ShareniuEventListener.java 自定义事件监听器

```
2 public class ShareniuEventListener implements ActivitiEventListener {
3     public void onEvent(ActivitiEvent event) {
4         System.err.println(event + ", " + event.getType());
5         switch (event.getType()) {
6             case VARIABLE_CREATED://变量创建事件
7                 ActivitiVariableEvent variableEvent = (ActivitiVariableEvent) event;
8                 break;
9             case ENTITY_CREATED://引擎创建事件
10                 ActivitiEventImpl activitiEventImpl = (ActivitiEventImpl) event;
11                 break;
12             }
13         }
14     public boolean isFailOnException() {
15         //出线异常程序是否继续执行
16         return false;
17     }
18 }
```

19 activiti.cfg.xml

```
20 <bean id="processEngineConfiguration"
21     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
22     <!-- 自定义的事件监听器 ShareniuEventListener 注入引擎 -->
23     <property name="eventListeners"ref="eventListeners"></property>
24     <!-- 注入事件类型为 TASK_CREATED, TASK_COMPLETED 的监听处理类,该属性是 Map 数
25     据结构 -->
26     <property name="typedEventListeners"><!-- 具体类型的事件监听器 -->
27     <map><entry key="TASK_CREATED, TASK_COMPLETED" value-ref="eventListeners"/>
28     </map>
29     </property>
30     <!-- 开启事件转发器,该属性默认为 true -->
31     <property name="enableEventDispatcher" value="true"></property>
32 </bean>
```

```

31 <!-- 自定义的事件监听器 ShareniuEventListener -->
32 <bean id="eventListeners" class="com.shareniu.chapter6.ShareniuEventListener"></bean>

```

在上述代码中,第2行定义的 ShareniuEventListener 类实现了 ActivitiEventListener 接口,第3~11行编写对应的监听事件处理逻辑,第20~32行主要完成自定义事件监听器添加以及将其注入流程引擎配置类。

注意

所有的事件类型均定义在 ActivitiEventType 类中,所有的事件处理类均可以通过 ActivitiEventBuilder 类的一系列方法来构造。

6.6.2 动态添加

对于动态添加事件监听器的操作来说,需要考虑如下。

(1) 因为事件监听器的注册和移除方法均定义在 ActivitiEventDispatcher 类中,所以对于动态添加或者移除事件监听器来说首先需要获取 ActivitiEventDispatcher 实例对象,由于 ProcessEngineConfigurationImpl 持有 ActivitiEventDispatcher 实例对象,因此该实例对象的获取也非常方便。

(2) 通过获取的 ActivitiEventDispatcher 实例对象进行事件监听器的注册或者移除工作,相关实现如代码清单 6-7 所示。

代码清单 6-7 App.java

```

1 ProcessEngine processEngine = null; // 获取到 Activiti ProcessEngine
2 ActivitiEventDispatcher eventDispatcher;
3 @Before
4 public void init() {
5     InputStream in =
6     App.class.getClassLoader().getResourceAsStream("com/shareniu/chapter6/activiti.cfg.xml");
7     ProcessEngineConfiguration pcf =
8     ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(in);
9     eventDispatcher = (ProcessEngineConfigurationImpl)
10     processEngine.getProcessEngineConfiguration().getEventDispatcher();
11 }
12 @Test
13 public void startProcessInstanceById() {
14     ShareniuEventListener listener = new ShareniuEventListener();
15     eventDispatcher.addEventListener(listener);
16     // 移除事件监听器
17     // eventDispatcher.removeEventListener(listener);
18 }

```

在上述代码中,第7~8行构造流程引擎配置类实例对象,然后第9~10行获取该实例对象中的 ActivitiEventDispatcher 实例对象,第14行实例化 ShareniuEventListener 类(该类的定义可以参考代码清单 6-6),第15~17行进行自定义事件监听器的添加和移除工作。

注意

动态添加的事件监听器,当程序或者引擎重启之后,该事件监听器会消失,除非再次执行添加操作。如果程序运行期间,通过该方式添加了过多的事件监听器,维护起来非常不方便,而通过配置方式添加的事件监听器会伴随流程引擎的初始化而初始化,建议通过配置方式添加事件监听器,更可靠也更便于维护。

6.7 日志监听器

在讲解日志监听器之前,先思考几个问题:为什么需要收集日志数据?收集什么样的日志数据?收集之后的日志数据又有什么用途?日志诞生的意义就是记录机器或者系统何时何地做了何事,耗费多长时间,接口有无异常等信息。根据日志记录开发人员可以很方便的排查问题、定位原因,以及观察服务接口是否健壮、性能是否稳定等,辅助开发人员找出业务系统可优化的方法和途径,因此日志数据的采集和分析非常重要,对于 Activiti 框架来说应该采集什么样的数据呢?使用数据采集的常见场景如下。

(1) 记录流程实例中任务节点的开始时间、完成时间、处理人、变量信息、持续时间以及该任务节点所属的流程文档。

(2) 记录流程实例的开始时间以及结束时间。

(3) 记录流程实例运转过程中附带的变量信息(一般键值对方式存在)。

(4) 记录流程实例途经的节点信息(流程实例运转轨迹),例如是按照流程文档模板循规蹈矩地运转还是频繁的进行节点之间的跳转。

(5) 记录是否存在卡顿的流程实例以及流程文档是否经常变动。

看到上述一系列的场景可能会有疑惑:直接从数据库查询相关数据并整理成期望的结果,就可以满足上述要求,为何需要日志记录?该处理方式确实可以,但试想一下如果流程引擎产生的数据量非常庞大且每次统计分析都要查询数据库,这样频繁操作数据库对数据库来说负担较大,而且更致命的一个问题是数据库中存储的部分数据是被引擎“消化”过滤的,并非第一数据来源,因此这样统计的数据可能有些偏差、不精确。上文讲解了自定义事件监听器转发事件的过程,何不以事件监听器为入口,对所有的事件进行监听并从事件处理类中获取期望的数据,进而从源头上解决日志采集工作,但如果对所有的事件进行监控并处理,可能显得力不从心,因为 Activiti 中定义的事件非常多,而且事件对应的处理类也不完全相同,所幸 Activiti 在 5.16 版本中增加了日志监听器来处理引擎产生的数据,从而使开发人员可以很方便的对日志进行收集、管理。接下来详细讲解日志监听器的相关知识。

6.7.1 初始化日志监听器

日志监听器的初始化过程如代码清单 6-8 所示。

代码清单 6-8 ProcessEngineConfigurationImpl.java

```
1 protected boolean enableDatabaseEventLogging = false;  
2 protected void initDatabaseEventLogging() {  
3     if (enableDatabaseEventLogging) {
```

```

4  getEventDispatcher().addEventListener(new EventLogger(clock, objectMapper));
5  }
6  }

```

initDatabaseEventLogging 方法主要负责初始化日志监听器,该方法的处理逻辑非常简单,首先第 3 行判断引擎配置类中的 enableDatabaseEventLogging 属性值是否为 true,如果是则第 4 行为事件转发器添加内置全局日志监听器 EventLogger。

通过分析上面的代码可以看出,开发人员要想开启日志记录功能,一是必须设置 ProcessEngineConfigurationImpl 类中的 enableDatabaseEventLogging 开关属性值为 true,二是必须开启事件转发功能,两者缺一不可。

构造 EventLogger 实例对象时必须传递 clock 参数(clock 类型,该类主要负责设置和获取时间)和 objectMapper 参数(objectMapper 类型,Activiti 默认使用 jackson 类库处理 JSON 格式数据,本书使用的程序包为 jackson-databind-2.3.0.jar)。

虽然 EventLogger 类作为全局日志监听器存在,但其设计不太合理,完全不符合 Activiti 中引擎配置类处理开关属性的一贯作风,这里直接实例化 EventLogger 类,如果开发人员不打算使用 EventLogger 类处理日志,就需要定义一个类,然后继承 StandaloneProcessEngineConfiguration 类并重写父类中的 initDatabaseEventLogging 方法,如果引擎在当前类中定义一个 ActivitiEventListener 类型的变量(开关属性)并根据该变量值执行不同的逻辑,岂不是更好。

注意

EventLogger 类本质上是一个全局事件监听器。

6.7.2 初始化日志处理器

new EventLogger(clock, objectMapper)操作主要用于实例化日志监听器,接下来详细分析该类的实例化处理逻辑,如代码清单 6-9 所示。

代码清单 6-9 EventLogger.java

```

1  protected Map<ActivitiEventType, Class> eventHandlers = new HashMap<ActivitiEventType,
   Class>();
2  public EventLogger() {
3      initializeDefaultHandlers(); }
4  public EventLogger(Clock clock, ObjectMapper objectMapper) {
5      this();
6      this.clock = clock;
7      this.objectMapper = objectMapper;
8  }
9  protected void initializeDefaultHandlers() {
10     addEventHandler(ActivitiEventType.TASK_CREATED, TaskCreatedEventHandler.class);
11     ...//省略一系列的日志处理器
12 }
13 void addEventHandler(ActivitiEventType eventType, Class eventHandlerClass) {

```

```

14     eventHandlers.put(eventType, eventHandlerClass);
15 }

```

上面代码的执行逻辑非常清晰：

(1) 第 4 行定义的构造方法首先会调用第 2 行定义的空参构造方法，然后初始化 clock 和 objectMapper 属性值。其中第 2 行定义的空参构造方法主要用于初始化一系列的事件处理类。

(2) initializeDefaultHandlers 方法负责将常用的事件以及该事件对应的日志处理器通过 addEventHandler 方法添加到 eventHandlers 集合中。

(3) 如果开发人员觉得 initializeDefaultHandlers 方法中初始化的事件不能满足业务日志收集需求，可以自定义一个日志监听器并继承 EventLogger 类，然后通过 addEventHandler 方法将期望处理的事件以及该事件对应的日志处理器添加到 eventHandlers 集合中即可。

6.7.3 日志处理器架构

下面分析日志处理器的架构，如图 6-2 所示。

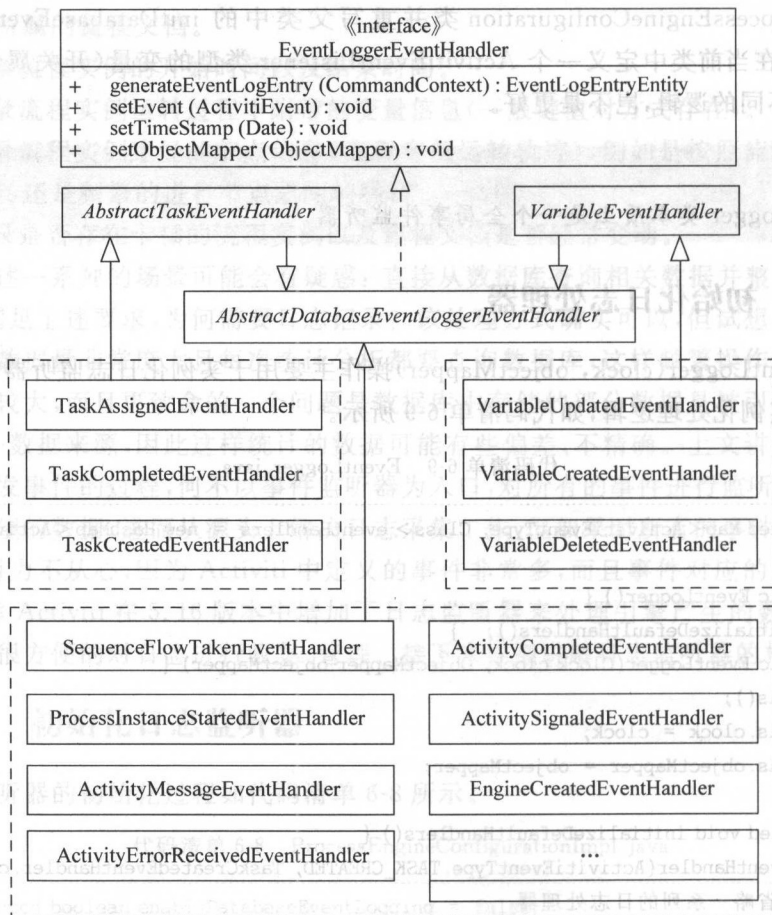


图 6-2 日志处理器架构图

(1) `EventListenerHandler`: 该接口定义了四个方法:

- ① `setEvent`, 该方法用于设置事件类型;
- ② `setTimeStamp`, 该方法用于设置事件发生的时间;
- ③ `setObjectMapper`, 该方法用于设置 `ObjectMapper` 实例对象;
- ④ `generateEventLogEntry`, 该方法用于生成 `EventLogEntryEntity` 实例对象 (`EventLogEntryEntity` 为 `ACT_EVT_LOG` 表对应的实体类)。

(2) `AbstractDatabaseEventListenerHandler`: 对接口 `EventListenerHandler` 中的方法进行实现。该类作为所有事件处理类的模板类存在, 全局统筹 `EventLogEntryEntity` 实体对象的生成工作。

(3) `AbstractTaskEventHandler`: 该抽象类继承 `AbstractDatabaseEventListenerHandler` 类的同时又作为所有与任务相关的事件处理类的父类, 并定义了模板方法 `handleCommonTaskFields`, 具体的实现类有三个: ① `TaskCreatedEventHandler` (创建任务); ② `TaskCompletedEventHandler` (完成任务); ③ `TaskAssignedEventHandler` (给任务分配处理人)。

(4) `VariableEventHandler`: 该抽象类继承 `AbstractDatabaseEventListenerHandler` 的同时又作为所有与变量操作相关的事件处理类的父类, 并定义了模板方法 `createData`, 具体的实现类有三个: ① `VariableCreatedEventHandler` (创建变量); ② `VariableUpdatedEventHandler` (更新变量); ③ `VariableDeletedEventHandler` (删除变量)。

6.7.4 收集日志数据入口

下面详细讲解日志监听器的转发事件方法 `onEvent`, 首先看一下这个方法的时序图 (如图 6-3 所示), `onEvent` 方法的相关实现如代码清单 6-10 所示。

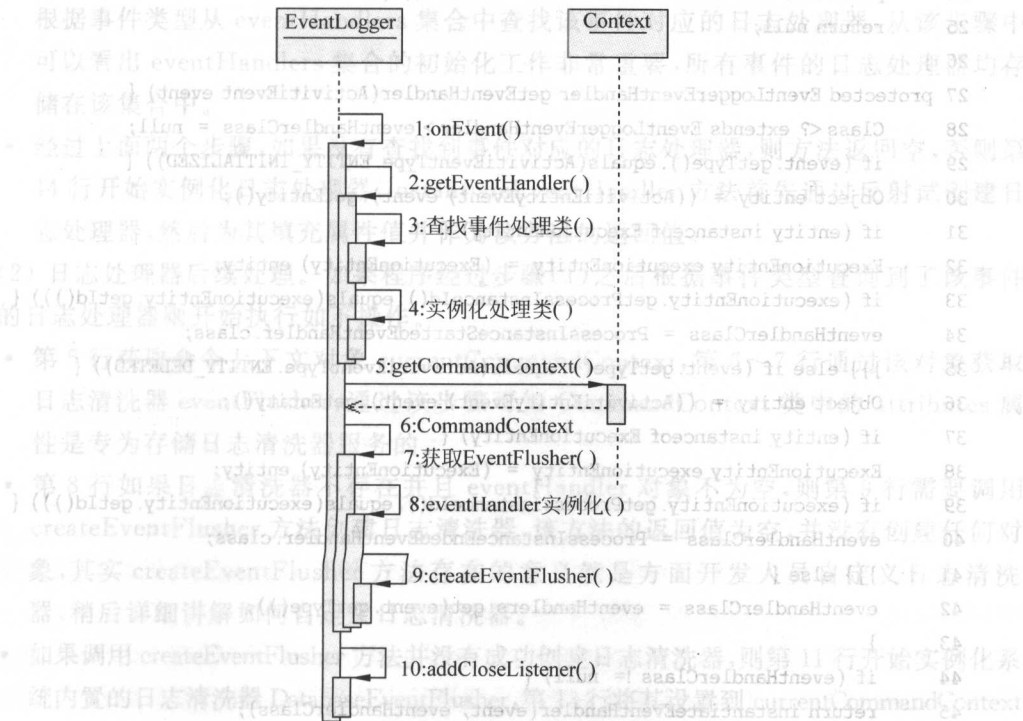


图 6-3 onEvent 方法执行时序图

代码清单 6-10 EventLogger.java

```

1 private static final String EVENT_FLUSHER_KEY = "eventFlusher";
2 public void onEvent(ActivitiEvent event) {
3     EventLoggerEventHandler eventHandler = getEventHandler(event);
4     if (eventHandler != null) {
5         CommandContext currentCommandContext = Context.getCommandContext();
6         EventFlusher eventFlusher = (EventFlusher)
7             currentCommandContext.getAttribute(EVENT_FLUSHER_KEY);
8         if (eventHandler != null && eventFlusher == null) {
9             eventFlusher = createEventFlusher();
10            if (eventFlusher == null) {
11                eventFlusher = new DatabaseEventFlusher(); // Default
12            }
13            currentCommandContext.addAttribute(EVENT_FLUSHER_KEY, eventFlusher);
14            currentCommandContext.addCloseListener(eventFlusher);
15            currentCommandContext.addCloseListener(new CommandContextCloseListener() {
16                public void closing(CommandContext commandContext) {}
17            });
18            public void closed(CommandContext commandContext) {
19                if (listeners != null) {
20                    for (EventLoggerListener listener : listeners) {
21                        listener.eventsAdded(EventLogger.this);
22                    }
23                }
24                eventFlusher.addEventHandler(eventHandler);
25                protected EventFlusher createEventFlusher() {
26                    return null;
27                }
28            protected EventLoggerEventHandler getEventHandler(ActivitiEvent event) {
29                Class<? extends EventLoggerEventHandler> eventHandlerClass = null;
30                if (event.getType().equals(ActivitiEventType.ENTITY_INITIALIZED)) {
31                    Object entity = ((ActivitiEntityEvent) event).getEntity();
32                    if (entity instanceof ExecutionEntity) {
33                        ExecutionEntity executionEntity = (ExecutionEntity) entity;
34                        if (executionEntity.getProcessInstanceId().equals(executionEntity.getId())) {
35                            eventHandlerClass = ProcessInstanceStartedEventHandler.class;
36                        } else if (event.getType().equals(ActivitiEventType.ENTITY_DELETED)) {
37                            Object entity = ((ActivitiEntityEvent) event).getEntity();
38                            if (entity instanceof ExecutionEntity) {
39                                ExecutionEntity executionEntity = (ExecutionEntity) entity;
40                                if (executionEntity.getProcessInstanceId().equals(executionEntity.getId())) {
41                                    eventHandlerClass = ProcessInstanceEndedEventHandler.class;
42                                }
43                            } else {
44                                eventHandlerClass = eventHandlers.get(event.getType());
45                            }
46                            if (eventHandlerClass != null) {
47                                return instantiateEventHandler(event, eventHandlerClass);
48                            }
49                        }
50                    }
51                }
52            }
53        }
54    }
55 }

```

```

47     return null;
48 }
49 protected EventLoggerEventHandler instantiateEventHandler(ActivitiEvent event,
50     Class<? extends EventLoggerEventHandler> eventHandlerClass) {
51     try {
52         EventLoggerEventHandler eventHandler = eventHandlerClass.newInstance();
53         eventHandler.setTimestamp(clock.getCurrentTime()); //获取当前时间
54         eventHandler.setEvent(event);
55         eventHandler.setObjectMapper(objectMapper);
56         return eventHandler;
57     } catch (Exception e) {}
58     return null;
59 }

```

结合上面的代码以及时序图将 onEvent 方法的处理流程梳理如下。

(1) 第 3 行调用 getEventHandler 方法获取事件对应的日志处理器,该方法的处理逻辑如下。

- 第 29~40 行首先判断事件类型,如果事件类型为 ENTITY_INITIALIZED 或者 ENTITY_DELETED,则获取该事件对应的日志处理器 ActivitiEntityEvent,并从该处理器中提取当前的实体对象 entity,接着判断 entity 对象是否为 ExecutionEntity 实例对象,如果是就需要对其进行转换并开始设置当前事件的日志处理器。
- 第 42 行如果事件类型不是 ENTITY_INITIALIZED 或者 ENTITY_DELETED,则根据事件类型从 eventHandlers 集合中查找该事件对应的日志处理器,从该步骤中可以看出 eventHandlers 集合的初始化工作非常重要,所有事件的日志处理器均存储在该集合中。
- 经过上面两个步骤,如果没有查找到事件对应的日志处理器,则方法返回空,否则第 44 行开始实例化日志处理器,instantiateEventHandler 方法首先通过反射试创建日志处理器,然后为其填充属性值并作为该方法的返回值。

(2) 日志处理器后续处理。如果程序经过步骤(1)之后根据事件类型查询到了该事件对应的日志处理器则开始执行如下操作。

- 第 5 行获取命令上下文对象 currentCommandContext,第 6~7 行通过该对象获取日志清洗器 eventFlusher,通过该步骤可知 CommandContext 类中的 attributes 属性是专为存储日志清洗器服务的。
- 第 8 行如果日志清洗器不存在并且 eventHandler 对象不为空,则第 9 行需要调用 createEventFlusher 方法创建日志清洗器,该方法的返回值为空,并没有创建任何对象,其实 createEventFlusher 方法存在的意义就是方便开发人员自定义日志清洗器,稍后详细讲解如何自定义日志清洗器。
- 如果调用 createEventFlusher 方法并没有成功创建日志清洗器,则第 11 行开始实例化系统内置的日志清洗器 DatabaseEventFlusher,第 13 行将其设置到 currentCommandContext 对象的 attributes 属性中,以方便程序后续获取。

- 上述一系列步骤执行完毕之后，第 14 行将日志清洗器 eventFlusher 对象添加到 currentCommandContext 对象中的 closeListeners 集合中，关于该集合的操作可以参考 15.11 节。
- 第 15~22 行使用了 Java 语言中的匿名类遍历当前类 EventLogger 中的 listeners 集合，并开始添加日志监听器，稍后详细讲解。
- 第 23 行通过 eventFlusher.addEventHandler(eventHandler)操作将事件对应的日志处理器注册到 EventFlusher 类中的 eventHandlers 集合中。

6.8 日志清洗器架构

6.8.1 数据库日志清洗器

上述 EventLogger 类中的 onEvent 方法的处理逻辑非常令人费解，该方法中并没有操作任何日志数据，那么引擎产生的日志数据格式是什么，又是如何处理的，是插入数据库还是简单的打印一下？下面重点分析 DatabaseEventFlusher 类是如何处理日志数据的，首先看一下该类的架构(如图 6-4 所示)。

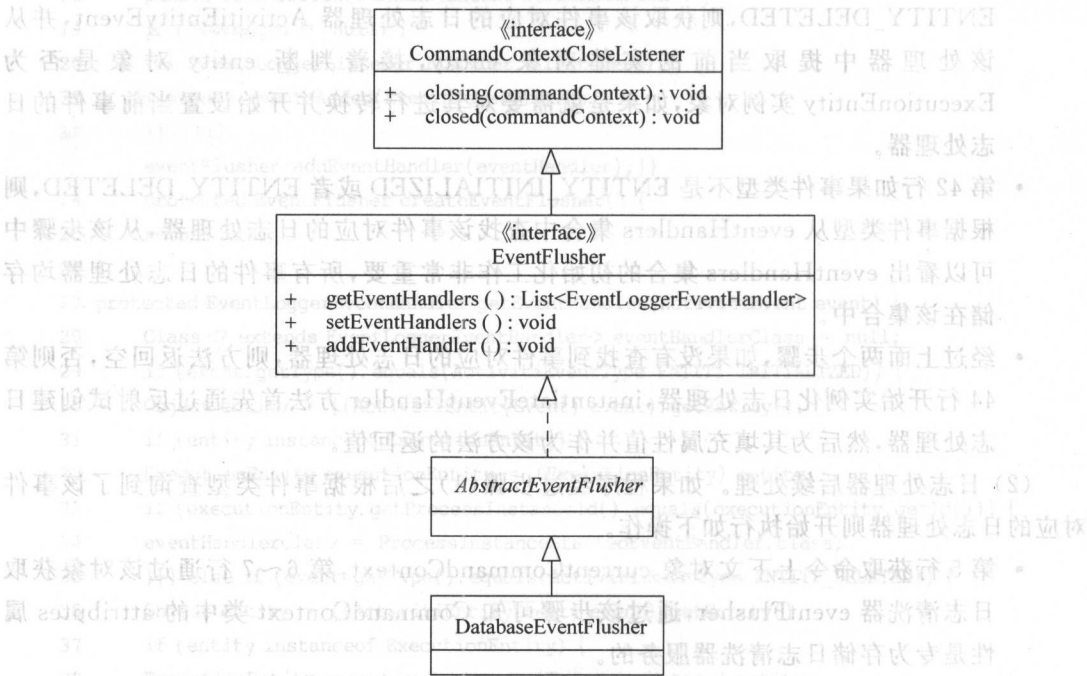


图 6-4 日志清洗器架构图

- (1) `CommandContextCloseListener`: 定义了两个方法，`closing`(命令上下文正在关闭中)和 `closed`(命令上下文已经关闭)。
- (2) `EventFlusher`: 继承 `CommandContextCloseListener` 接口，在 `CommandContextCloseListener` 接口定义功能的基础上增加了获取日志处理器、添加日志处理器的支持。
- (3) `AbstractEventFlusher`: 对 `EventFlusher` 接口中的部分方法进行实现。

(4) DatabaseEventFlusher: 继承 AbstractEventFlusher 类, 并对 CommandContext-CloseListener 接口中定义的 closing 方法进行实现。

了解了以上每个类的职责之后, 接下来详细分析 DatabaseEventFlusher 是如何处理日志数据的, 该类的相关定义如代码清单 6-11 所示。

代码清单 6-11 DatabaseEventFlusher.java

```
1 public void closing(CommandContext c) {
2     EventLogEntryEntityManager elem = c.getEventLogEntryEntityManager();
3     for (EventLoggerEventHandler eventHandler : eventHandlers) {
4         try {
5             elem.insert(eventHandler.generateEventLogEntry(commandContext));
6         } catch (Exception e) {}
7     }
8 }
```

closing 方法完成了日志数据的入库操作, 具体步骤以及实现功能总结如下。

(1) 第 2 行根据 CommandContext 实例对象获取事件日志实体管理类 elem 对象。EventLogEntryEntityManager 类负责 ACT_EVT_LOG 表的操作。

(2) 第 3~7 行循环遍历 eventHandlers 集合, 第 5 行通过 eventHandler.generateEventLogEntry(commandContext) 方法生成 EventLogEntryEntity 实例对象, 最终通过调用 elem 对象中的 insert 方法将日志数据添加到会话缓存。

6.8.2 生成日志数据

通过分析上面代码的处理逻辑, 可能会有这样的疑问: 日志数据是如何生成的? DatabaseEventFlusher 类中的 closing 方法在何种场景下被调用? 接下来具体分析生成日志数据的过程, 由于日志数据的格式比较类似, 仅仅是数据不同而已, 所以这里以 TaskCreatedEventHandler 类为例详细探讨其内部实现机制, 如代码清单 6-12 所示。

代码清单 6-12 TaskCreatedEventHandler.java

```
1 public EventLogEntryEntity generateEventLogEntry(CommandContext commandContext) {
2     TaskEntity task = (TaskEntity) ((ActivitiEntityEvent) event).getEntity();
3     Map<String, Object> data = handleCommonTaskFields(task);
4     return createEventLogEntry(task.getProcessDefinitionId(), task.getProcessInstanceId(),
5     task.getExecutionId(), task.getId(), data);
6 }
```

上面代码的处理逻辑如下所示。

(1) 第 2 行获取 TaskEntity 实例对象。

(2) 第 3 行调用 handleCommonTaskFields 方法封装通用字段。

(3) 第 4~5 行委托父类的 createEventLogEntry 方法生成日志数据。

由于当前类中并没有 handleCommonTaskFields 和 createEventLogEntry 方法, 因此需要跟进 TaskCreatedEventHandler 类的父类 AbstractTaskEventHandler, 相关实现如代码

清单 6-13 所示。

代码清单 6-13 AbstractTaskEventHandler.java 和 AbstractDatabaseEventLoggerEventHandler.java

```

1  AbstractTaskEventHandler.java:
2  //handleCommonTaskFields 方法只服务于任务相关的事件,所以定义在 AbstractTaskEventHandler 类中
3  protected Map<String, Object> handleCommonTaskFields(TaskEntity task) {
4      Map<String, Object> data = new HashMap<String, Object>();
5      //实例化 data 集合作为承载日志数据的容器
6      putInMapIfNotNull(data, Fields.ID, task.getId()); //将一系列的数据添加到集合中
7      ...//省略一系列的属性判断代码
8      return data;
9  }
10 //该方法用于过滤 map 集合的数据
11 public void putInMapIfNotNull(Map<String, Object> map, String key, Object value) {
12     if (value != null) {
13         map.put(key, value);
14     }
15 }
16 AbstractDatabaseEventLoggerEventHandler.java:
17 //该方法主要为 EventLogEntryEntity 实例填充属性
18 protected EventLogEntryEntity createEventLogEntry(String type,
19     String processDefinitionId, String processInstanceId, String executionId,
20     String taskId, Map<String, Object> data) {
21     EventLogEntryEntity eventLogEntry = new EventLogEntryEntity();
22     //为对象填充一系列的属性
23     ...//省略属性填充
24     //该方法直接将 data 数据转化为 byte
25     eventLogEntry.setData(objectMapper.writeValueAsBytes(data));
26     return eventLogEntry;
27 }

```

在上述代码中,第 4~7 行根据 task 参数初始化 data 集合,其中第 5 行调用第 10 行定义的 putInMapIfNotNull 方法,该方法主要是对空数据进行过滤。第 20~23 行实例化 EventLogEntryEntity 类并为其填充属性。

6.8.3 日志存储

DatabaseEventFlusher 类中的 closing 方法在何种场景下被调用呢? 因为该操作涉及数据入库,数据的入库操作在 CommandContext 类中的 close 方法中完成,可以参考第 15 章。

6.9 自定义日志清洗器

由于 Activiti 将所有的日志数据插入到 ACT_EVT_LOG 表中,可能在实际项目开发中,开发人员更青睐使用日志处理框架进行日志数据的采集和分析,比如 Kafka 框架,该框架是一个分布式的、可划分的、冗余备份的持久性日志处理框架,主要用于处理活跃的流式

数据。当然也可以将所有的日志数据交给 Elasticsearch 搜索引擎进行管理,这个时候如果使用 Activiti 默认的存储方式(数据库)存储日志数据可能就力不从心了,因此必须想一个合理可行的办法替换引擎默认的日志清洗器,相关实现可以分为如下几个步骤。

(1) 自定义一个日志清洗器,如代码清单 6-14 所示。

代码清单 6-14 KafkaEventFlusher.java

```
1 //自定义日志清洗器
2 public class KafkaEventFlusher extends AbstractEventFlusher {
3     public void closing(CommandContext c) { //自定义日志清洗器
4         EventLogEntryEntityManager eventLogEntryEntityManager = c.getEventLogEntryEntityManager();
5         for (EventLoggerEventHandler eventHandler : eventHandlers) {
6             EventLogEntryEntity generateEventLogEntry = eventHandler.generateEventLogEntry(c);
7             //TODO 将获取之后的日志实体转化为符合业务的实体对象
8             // eventLogEntryEntityManager.insert(eventHandler.generateEventLogEntry(c));
9         }
10    }
11 }
```

在上述代码中,第 4 行获取 EventLogEntryEntityManager 实例对象,第 5 行循环遍历 eventHandlers 集合,第 6 行获取 EventLogEntryEntity 实例对象。KafkaEventFlusher 类用来处理日志数据,可以将日志数据插入数据库、搜索引擎,也可以传递给日志系统。日志数据的存储方式可以结合业务进行选用。

(2) 自定义日志监听器如代码清单 6-15 所示。

代码清单 6-15 ShareniuEventLogger.java

```
1 public class ShareniuEventLogger extends EventLogger{
2     //通过继承 EventLogger 类并重写 createEventFlusher 方法
3     protected EventFlusher createEventFlusher() {
4         KafkaEventFlusher kafkaEventFlusher = new KafkaEventFlusher();
5         return kafkaEventFlusher;
6     }
7     public ShareniuEventLogger(Clock clock, ObjectMapper objectMapper) {
8         super();
9         this.clock = clock;
10        this.objectMapper = objectMapper;
11    }
12    protected void initializeDefaultHandlers() {
13        super.initializeDefaultHandlers();
14        addEventHandler(ActivitiEventType.ENTITY_UPDATED, TaskUpdatedEventHandler.class);
15    }
16 }
```

在上述代码中,第 3~6 行重写 EventLogger 类中的 createEventFlusher 方法,其中第 4 行实例化 KafkaEventFlusher 类,第 7~11 行初始化 ShareniuEventLogger 类中的属性,第 12~15 行重写 EventLogger 类中的 initializeDefaultHandlers 方法,其中第 13 行调用父类的

initializeDefaultHandlers 方法,第 14 行添加自定义的日志处理器 TaskUpdatedEventHandler 类,该类的定义如代码清单 6-16 所示。

代码清单 6-16 TaskUpdatedEventHandler.java

```
1 public class TaskUpdatedEventHandler extends AbstractTaskEventHandler {
2     public EventLogEntryEntity generateEventLogEntry(CommandContext commandContext){
3         return null; //可以自行实现
4     }
5 }
```

在上述代码中,开发人员可以在第 2 行定义的 generateEventLogEntry 方法中实现自定义业务逻辑。

(3) 由于引擎默认使用的是 EventLogger 处理日志数据,且没有给开发人员预留扩展的余地,因此需要自定义一个引擎配置类,如代码清单 6-17 所示。

代码清单 6-17 MyStandaloneProcessEngineConfiguration.java

```
1 public class MyStandaloneProcessEngineConfiguration extends
2     StandaloneProcessEngineConfiguration {
3     protected void initDatabaseEventLogging() {
4         if (enableDatabaseEventLogging) {
5             getEventDispatcher().addEventListener( new ShareniuEventLogger(clock, objectMapper));
6         }
7     }
```

在上述代码中,MyStandaloneProcessEngineConfiguration 类继承 StandaloneProcessEngineConfiguration 类,并重写父类中的 initDatabaseEventLogging 方法,第 4~5 行将 ShareniuEventLogger 类作为默认日志监听器。

(4) activiti.cfg.xml 配置文件如代码清单 6-18 所示。

代码清单 6-18 activiti.cfg.xml

```
1 <bean id="processEngineConfiguration"
2     class="com.shareniu.chapter6.MyStandaloneProcessEngineConfiguration">
3     //需要设置 enableDatabaseEventLogging 参数值为 true,否则无法进行日志事件的监听。
4     <property name="enableDatabaseEventLogging" value="true"></property>
5 </bean>
```

在上述代码中,第 4 行在配置文件中开启日志监听功能。至此自定义日志清洗器讲解完毕,可结合该案例自行实现和测试。

6.9 自定义日志清洗器

第7章

流程文档部署原理

前面章节中详细讲解了流程文档的部署操作以及元素解析,那么引擎将流程文档中的元素解析之后又做了什么操作呢?在实际项目开发中,如果期望在流程文档部署动作前后做一系列的操作,又该如何实现呢?如果觉得 Activiti 提供的默认部署器限制太多,又该如何扩展呢?带着这些问题开始深入学习流程文档部署的内部实现机制。

7.1 初始化部署器

前面章节中讲解了流程文档部署的相关工作,下面分析流程引擎收到部署资源的命令之后如何处理。这里以部署器的初始化过程为切入口,探究其内部实现机制,部署器的初始化入口位于 ProcessEngineConfigurationImpl 类中的 initDeployers 方法,该方法的具体实现如代码清单 7-1 所示。

代码清单 7-1 ProcessEngineConfigurationImpl.java

```
1 protected void initDeployers() {
2     //首先判断 deployers 集合是否已经初始化,如果该集合已经初始化则不会进行实例化操作
3     if (this.deployers == null) {
4         //初始化 deployers 集合
5         this.deployers = new ArrayList<Deployer>();
6         //如果前置部署器不为空,则向集合中添加该值
7         if (customPreDeployers != null) {
8             this.deployers.addAll(customPreDeployers);
9         }
10        //添加内置部署器
11        this.deployers.addAll(getDefaultDeployers());
12        //如果后置部署器不为空,向集合中添加该值
13        if (customPostDeployers != null) {
14            this.deployers.addAll(customPostDeployers);
15        }
16    }
17 }
```



```

15     }
16 }
17 if (deploymentManager == null) {
18     //实例化部署管理器
19     deploymentManager = new DeploymentManager();
20     //向部署管理器中添加部署器集合
21     deploymentManager.setDeployers(deployers);
22     //如果流程定义缓存限制值为空则执行不同的代码逻辑
23     //processDefinitionCacheLimit 值默认是 -1 表示不开启缓存
24     if (processDefinitionCache == null) {
25         if (processDefinitionCacheLimit <= 0) {
26             processDefinitionCache = new DefaultDeploymentCache<ProcessDefinitionEntity>();
27         } else {
28             processDefinitionCache = new DefaultDeploymentCache<>(processDefinitionCacheLimit);
29         }
30         ...//省略不同的缓存处理类初始化逻辑
31     }
32     ...//省略设置其他的缓存处理类
33     deploymentManager.setProcessDefinitionCache(processDefinitionCache);
34 }
35 }

```

initDeployers 方法的处理过程比较复杂,在深入讲解每一个实现细节之前,先总览整个方法。

(1) 第 3 行,如果 deployers 开关属性为空,则执行第 5~8 行初始化 deployers 集合。

(2) 初始化部署器集合:第 8 行添加前置部署器,第 11 行添加内置部署器,第 14 行添加后置部署器。其中开发人员可以扩展的部署器包含前置部署器 customPreDeployers 和后置部署器 customPostDeployers,为什么需要使用两个变量分别定义前置部署器和后置部署器,定义一个部署器变量岂不是更好吗?其实不然,这样设计的好处是因为部署器集合是 List 数据结构,List 数据结构是有序的,也是为了后续遍历该集合时有先后顺序之分。

(3) 第 17 行,如果 deploymentManager 开关属性为空,则执行如下逻辑。

第 19 行实例化 DeploymentManager 类,第 24 行判断 processDefinitionCache 开关属性是否为空,如果该属性值为空,则第 25 行判断 processDefinitionCacheLimit 开关属性是否小于或者等于 0,如果该值小于或者等于 0,则第 26 行实例化 DefaultDeploymentCache 类,否则第 28 行实例化 DefaultDeploymentCache 类并传入该类构造方法需要的输入参数 processDefinitionCacheLimit,第 33 行将 processDefinitionCache 设置到 deploymentManager 对象中。

约定

本书如果没有特殊说明,则部署管理器均为 DeploymentManager 类,部署器均为 BpmnDeployer 类。

关于 DeploymentManager 类的职责以及缓存使用可以参考第 8.3 节。在深入学习部署管理器之前,首先了解一下内置部署器的初始化过程。

7.1.1 初始化内置部署器

上面详细讲解了部署器的初始化过程,下面分析内置部署器的初始化过程,如果开发人员没有配置 deployers 开关属性值,则 getDefaultDeployers() 方法必定会被调用,该方法主要用于获取系统内置部署器,具体实现如代码清单 7-2 所示。

代码清单 7-2 ProcessEngineConfigurationImpl.java

```
1 protected Collection getDefaultDeployers() {
2     //初始化 defaultDeployers 集合,用来存储所有的部署器实例对象
3     List<Deployer> defaultDeployers = new ArrayList<Deployer>();
4     if (bpmnDeployer == null) {
5         bpmnDeployer = new BpmnDeployer(); //实例化 BpmnDeployer 类
6     }
7     bpmnDeployer.setExpressionManager(expressionManager); //设置表达式管理器
8     bpmnDeployer.setIdGenerator(idGenerator); //设置 Id 生成器
9     if (bpmnParseFactory == null) {
10        //设置默认的 BPMN 解析工厂,该工厂负责创建 BpmnParse 对象
11        bpmnParseFactory = new DefaultBpmnParseFactory(); }
12    if (activityBehaviorFactory == null) { //初始化 activityBehaviorFactory
13        DefaultActivityBehaviorFactory defaultActivityBehaviorFactory =
14            new DefaultActivityBehaviorFactory();
15        defaultActivityBehaviorFactory.setExpressionManager(expressionManager);
16        activityBehaviorFactory = defaultActivityBehaviorFactory;
17    } else if ((activityBehaviorFactory instanceof AbstractBehaviorFactory)
18        && ((AbstractBehaviorFactory) activityBehaviorFactory).getExpressionManager() == null) {
19        ((AbstractBehaviorFactory)
20            activityBehaviorFactory).setExpressionManager(expressionManager);
21    }
22    ...//省略 listenerFactory 开关属性的判断和操作
23    if (bpmnParser == null) {
24        bpmnParser = new BpmnParser(); //实例化 BpmnParser 类
25    }
26    bpmnParser.setExpressionManager(expressionManager);
27    bpmnParser.setBpmnParseFactory(bpmnParseFactory);
28    bpmnParser.setActivityBehaviorFactory(activityBehaviorFactory);
29    bpmnParser.setListenerFactory(listenerFactory);
30    List<BpmnParseHandler> parseHandlers = new ArrayList<BpmnParseHandler>();
31    if (getPreBpmnParseHandlers() != null) {
32        parseHandlers.addAll(getPreBpmnParseHandlers()); //添加前置对象解析器
33    }
34    parseHandlers.addAll(getDefaultBpmnParseHandlers()); //添加内置对象解析器
35    if (getPostBpmnParseHandlers() != null) {
36        parseHandlers.addAll(getPostBpmnParseHandlers()); //添加后置对象解析器
37    }
38    BpmnParseHandlers bpmnParseHandlers = new BpmnParseHandlers();
39    bpmnParseHandlers.addHandlers(parseHandlers);
40    bpmnParser.setBpmnParserHandlers(bpmnParseHandlers);
41    bpmnDeployer.setBpmnParser(bpmnParser);
```

```

42 defaultDeployers.add(bpmnDeployer);
43 return defaultDeployers;
44 }

```

上述代码的处理逻辑如下。

(1) 实例化 BpmnDeployer 类。

首先第 4 行判断 bpmnDeployer 开关属性值是否为空,如果该属性值为空,则第 5 行实例化 BpmnDeployer 类。第 7~8 行填充 bpmnDeployer 对象的各种属性值,包括表达式管理器对象 expressionManager 和 id 生成器对象 idGenerator,这两个对象的初始化工作分别在 ProcessEngineConfigurationImpl 类的 initExpressionManager() 和 initIdGenerator() 方法中完成。表达式管理器的主要工作就是解析 Juel 表达式形如 `${shareniu}`,id 生成器的作用就是负责所有的数据库主键 id 值生成。

(2) 第 9 行判断 bpmnParseFactory 开关属性值是否为空,如果该属性值为空,那么第 11 行实例化 DefaultBpmnParseFactory 类,该类负责创建 BpmnParse 实例对象。换言之,BpmnParse 实例对象的创建工作,开发人员也可以自定义实现。

(3) 实例化活动行为工厂类 ActivityBehaviorFactory。

第 12 行判断 activityBehaviorFactory 开关属性值是否为空,如果该属性值为空,则第 13~14 行实例化 DefaultActivityBehaviorFactory 类,第 15 行为其填充表达式管理器属性值;如果该属性值不为空,则第 17~18 行需要判断 activityBehaviorFactory 对象是否为 AbstractBehaviorFactory 实例对象,如果是并且该对象中表达式管理器为空,则第 19~20 行为其填充表达式管理器属性值。

(4) 第 22 行实例化监听器工厂类 ListenerFactory。可以参考步骤上述的处理过程。

(5) 实例化 BpmnParser 类。

(6) 第 23 行判断 bpmnParser 开关属性值是否为空,如果该属性值为空,则第 24 行直接实例化 BpmnParser 类,该类非常重要,负责创建 BpmnParse 实例对象,而 BpmnParse 类负责全局调度元素解析和对象解析工作,BpmnParser 类的重要性不言而喻。

(7) 第 26~29 行填充 bpmnParser 对象的属性值,填充的属性有 expressionManager、bpmnParseFactory、activityBehaviorFactory、listenerFactory。

(8) 初始化 parseHandlers 集合。

(9) 第 31~33 行获取前置对象解析器集合,然后第 34 行获取系统内置对象解析器集合,最后第 35~37 行获取后置对象解析器集合。对象解析器可以参考第 13 章。

(10) 第 38 行实例化 BpmnParseHandlers 类,然后第 39 行将 parseHandlers 集合赋值到 bpmnParseHandlers 对象中。

(11) 第 40 行将 bpmnParseHandlers 对象作为属性赋值到 bpmnParser 对象中。

(12) 第 41 行将 bpmnParser 对象作为属性值赋值到 bpmnDeployer 对象中。

说明

DefaultActivityBehaviorFactory 类主要负责流程三大要素等行为类的创建工作,DefaultListenerFactory 类主要负责创建任务监听器、执行监听器和事件监听器。

7.1.2 部署器依赖关系

getDefaultDeployers 方法是获取内置部署器的入口,该方法涉及部署器的实例化、元素解析器的实例化、对象解析器的实例化以及相关属性填充等操作,接下来讲解上述每个类的功能职责,以辅助理解该方法的处理逻辑,该方法所涉及的类之间的相互依赖关系如图 7-1 所示。

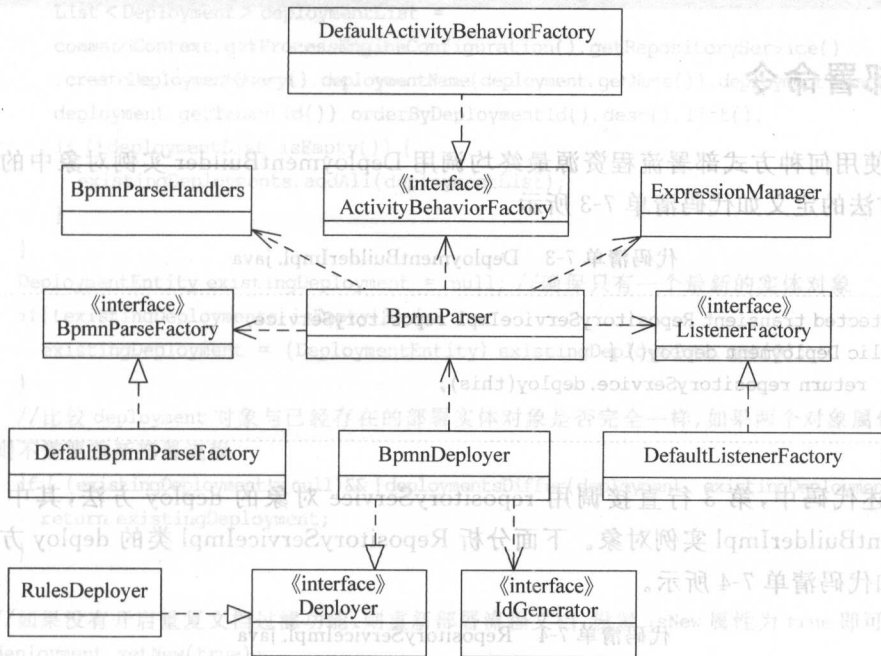


图 7-1 getDefaultDeployers 方法涉及的类与类之间的关系

接下来先简单介绍每个类的职责,后续章节中会逐渐深入讲解。

- (1) Deployer 接口: 定义了部署流程资源的方法,所有的部署器均需要实现该接口。
- (2) BpmnDeployer: 默认 Bpmn 部署器。
- (3) RulesDeployer: 规则部署器,该类只有一个 deploy 方法,其内部使用 KnowledgeBuilder 类查找并编译规则文件。规则文件通常使用 .drl 作为后缀。
- (4) ExpressionManager: 表达式管理器,主要用来解析流程文档中定义的 Juel 表达式。
- (5) IdGenerator: id 生成器,负责生成引擎中所有需要使用到的数据库主键 id 值。
- (6) ActivityBehaviorFactory: 该接口定义了一系列创建活动行为类的方法。
- (7) ListenerFactory: 该接口定义了一系列创建任务监听器、执行监听器和事件监听器的方法。
- (8) BpmnParseFactory: 负责创建 BpmnParse 实例对象。
- (9) DefaultBpmnParseFactory: BpmnParseFactory 接口的默认实现类。
- (10) BpmnParseHandler: 定义了将 BaseElement 实例对象转化为 ActivityImpl 实例对象或者 TransitionImpl 实例对象的方法以及根据 BaseElement 实例的具体类型查找其对应

应的对象解析器方法。

(11) BpmnParseHandlers: 解析对象的入口, 该类定义了解析对象的方法, 内部持有 BpmnParseHandler 类型的集合, 这样流程引擎解析对象时就可以从 BpmnParseHandler 集合中查找到该对象对应的所有解析器, 从而完成对象解析工作。

注意

KnowledgeBuilder 类位于 knowledge-api-5.5.0.Final.jar 中。

7.2 部署命令

不管使用何种方式部署流程资源最终均调用 DeploymentBuilder 实例对象中的 deploy 方法, 该方法的定义如代码清单 7-3 所示。

代码清单 7-3 DeploymentBuilderImpl.java

```
1 protected transient RepositoryServiceImpl repositoryService;
2 public Deployment deploy() {
3     return repositoryService.deploy(this);
4 }
```

在上述代码中, 第 3 行直接调用 repositoryService 对象的 deploy 方法, 其中 this 为 DeploymentBuilderImpl 实例对象。下面分析 RepositoryServiceImpl 类的 deploy 方法的处理逻辑, 如代码清单 7-4 所示。

代码清单 7-4 RepositoryServiceImpl.java

```
1 public Deployment deploy(DeploymentBuilderImpl deploymentBuilder) {
2     return commandExecutor.execute(new DeployCmd<Deployment>>(deploymentBuilder));
3 }
```

在上述代码中, 第 2 行首先实例化 DeployCmd 命令类, 然后使用 commandExecutor 对象执行该命令类。关于命令的执行过程可以参考第 12 章, 这里暂时将关注点放到流程部署命令 DeployCmd 类的处理逻辑中, 该类核心定义如代码清单 7-5 所示。

代码清单 7-5 DeployCmd.java

```
1 public Deployment execute(CommandContext commandContext) {
2     //获取 DeploymentEntity 实例对象
3     DeploymentEntity deployment = deploymentBuilder.getDeployment();
4     ... //省略设置部署时间为系统当前时间
5     if (deploymentBuilder.isDuplicateFilterEnabled()) { //判断是否开启了过滤重复文档功能
6         List<Deployment> existingDeployments = new ArrayList<Deployment>();
7         //如果 deployment 对象的租户 tenantId 值不存在
8         if (deployment.getTenantId() == null || "".equals(deployment.getTenantId())) {
9             //根据 deployment 对象的 name 值查找 act_re_deployment 表中是否已经存在记录
10            DeploymentEntity existingDeployment = commandContext.getDeploymentEntityManager()
```

```
11 .findLatestDeploymentByName(deployment.getName());
12 if (existingDeployment != null) {
13     existingDeployments.add(existingDeployment);
14 }
15 } else {
16     //根据租户 tenantId 和 name 值从数据库 act_re_deployment 表中查找数据,如果存在则
    添加到集合
17     List<Deployment> deploymentList =
18     commandContext.getProcessEngineConfiguration().getRepositoryService()
19     .createDeploymentQuery().deploymentName(deployment.getName()).deploymentTenantId(
20     deployment.getTenantId()).orderByDeploymentId().desc().list();
21     if (!deploymentList.isEmpty()) {
22         existingDeployments.addAll(deploymentList);
23     }
24 }
25 DeploymentEntity existingDeployment = null; //确保只有一个最新的实体对象
26 if(!existingDeployments.isEmpty()) {
27     existingDeployment = (DeploymentEntity) existingDeployments.get(0);
28 }
29 //比较 deployment 对象与已经存在的部署实体对象是否完全一样,如果两个对象属性值完全
    一致,则不需要重新部署流程
30 if ( (existingDeployment!= null)&& !deploymentsDiffer(deployment, existingDeployment)) {
31     return existingDeployment;
32 }
33 }
34 //如果没有开启重复文档过滤功能,则重新部署流程文档,设置 isNew 属性为 true 即可
35 deployment.setNew(true);
36 //将部署实体对象添加到会话缓存中,后续插入数据库
37 commandContext.getDeploymentEntityManager().insertDeployment(deployment);
38 ...// 省略如果开启事件转发机制,则转发 ENTITY_CREATED 事件
39 //设置是否开启 BPMN2.0 XSD 文件验证和流程文档格式验证
40 Map<String, Object> deploymentSettings = new HashMap<String, Object>();
41 deploymentSettings.put(DeploymentSettings.IS_BPMN20_XSD_VALIDATION_ENABLED,
42 deploymentBuilder.isBpmn20XsdValidationEnabled());
43 deploymentSettings.put(DeploymentSettings.IS_PROCESS_VALIDATION_ENABLED,
44 deploymentBuilder.isProcessValidationEnabled());
45 //真正开始部署流程文档
46 commandContext.getProcessEngineConfiguration().getDeploymentManager().deploy(deployment,
47 deploymentSettings);
48 //如果设置流程开启事件,则进行相应的处理
49 if (deploymentBuilder.getProcessDefinitionsActivationDate() != null) {
50     scheduleProcessDefinitionActivation(commandContext, deployment);
51 }
52 ...// 省略如果开启事件转发机制,则转发 ENTITY_INITIALIZED 事件
53 return deployment;
54 }
```

以上便是流程文档部署的全过程,在细化讲解之前,首先看该方法的时序图,如图 7-2 所示。

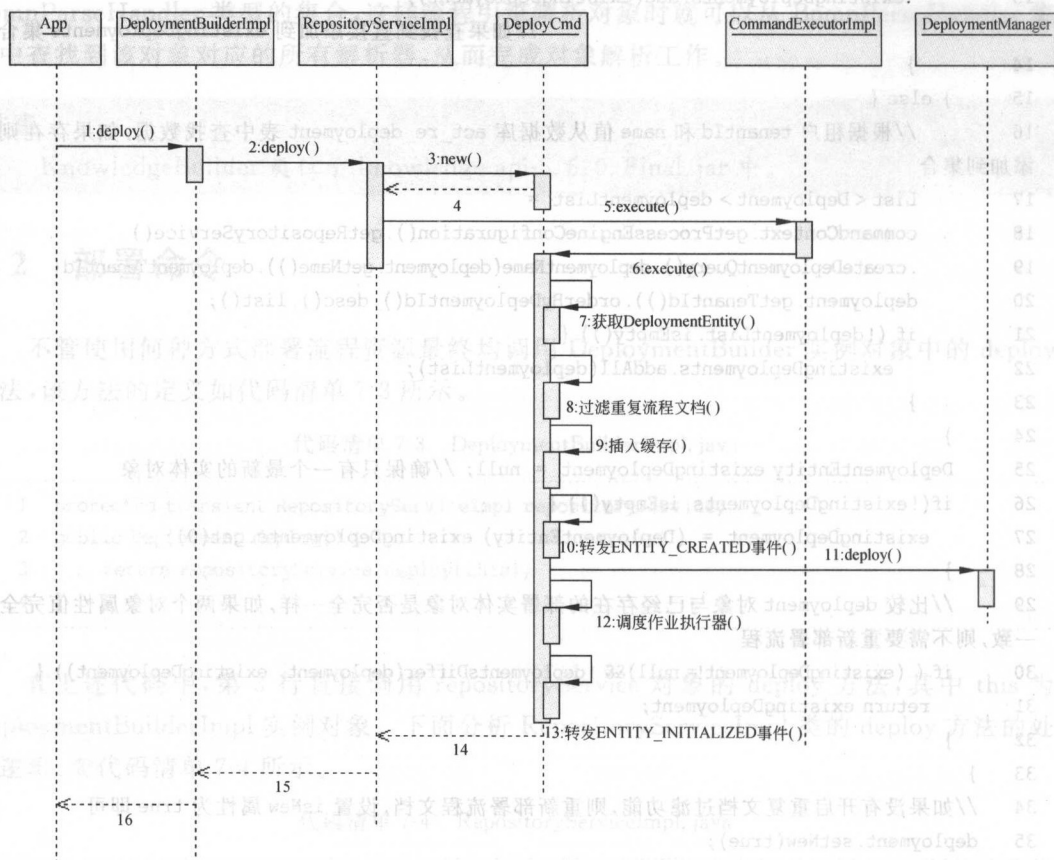


图 7-2 流程文档部署时序图

结合上述代码和时序图,分析 execute 方法的处理逻辑并将其梳理总结如下。

(1) 第 5 行判断是否需要过滤重复流程文档。

如果开发人员开启了过滤重复流程文档功能,则执行第 6~33 行验证数据库中是否已经存在该流程文档,如果数据库中已经存在,则不会重复部署。

(2) 第 35 行设置标识位。

设置 deployment 对象的 isNew 属性值为 true,isNew 参数值表示该流程文档是否可以部署操作,true 表示可以进行部署操作。

(3) 插入会话缓存如第 37 行所示。

将 deployment 对象添加到会话缓存中,后续可以将其直接插入数据库相应的表中。

(4) 第 40~44 行设置流程文档验证规则。

根据客户端的配置信息设置是否开启 BPMN2.0 XSD 文件验证和流程文档验证功能,流程引擎默认开启了这两种验证功能,禁用流程文档验证功能形如 repositoryService.createDeployment().disableBpmnValidation(),禁用 BPMN2.0 XSD 文件验证功能形如 repositoryService.createDeployment().disableSchemaValidation()。

(5) 第 46~47 行部署流程文档。

经过上面一系列的验证和赋值操作,终于开始部署流程文档。

(6) 第 50 行激活调度器。

如果开发人员部署流程文档的时候明确指定了该流程的激活时间,则需要执行第 50 行操作,该功能的开启形如 repositoryService.createDeployment().activateProcessDefinitionsOn(date),该操作主要针对挂起流程、激活流程。

至此 Activiti 引擎部署流程文档的处理流程已经总览完毕,该处理过程中看到了 deployment 对象添加到会话缓存以及部署资源方法的调用痕迹,接下来细化讲解该过程中涉及的每个实现细节。

7.2.1 过滤重复文档

如果开发人员设置了 DeploymentBuilderImpl 实例对象的 isDuplicateFilterEnabled 属性值为 true,则部署流程文档之前需要验证该流程文档是否已经存在于数据库中。

该功能的开启形如 repositoryService.createDeployment().enableDuplicateFiltering()。下面详细分析 Activiti 是如何过滤重复文档的。

(1) 第 8 行获取并判断 deployment 对象中的 tenantId(租户标识)属性值,如果该属性值不存在或者为空,则执行第 10~14 行代码委托 DeploymentEntityManager 实例对象中的 findLatestDeploymentByName 方法从 ACT_RE_DEPLOYMENT 表中查询数据,否则第 17~23 行以 deployment 对象中的 name 值和 tenantId 作为查询条件从 ACT_RE_DEPLOYMENT 表中查询数据,最后将查询结果添加到 existingDeployments 集合中。其中,findLatestDeploymentByName(String deploymentName)方法的具体实现如代码清单 7-6 所示。

代码清单 7-6 DeploymentEntityManager.java

```
1 public DeploymentEntity findLatestDeploymentByName(String deploymentName) {
2     //调用 SqlSession 实例中的 selectList(String statement, Object parameter) 方法并传入 SQL 语
    句需要的参数值
3     List<?> list = getDbSqlSession().selectList("selectDeploymentsByName", deploymentName, 0, 1);
4     if (list != null && !list.isEmpty()) {                //如果查询到数据则只返回第一条数据
5         return (DeploymentEntity) list.get(0);
6     }
7     return null;                                          //默认返回 null
8 }
```

在上述代码中,第 3 行需要执行的 SQL 语句对应 Deployment.xml 文件中 id 值为 selectDeploymentsByName 的查询语句(有关 SQL 语句的封装和执行过程可以参考第 15 章),如代码清单 7-7 所示。

代码清单 7-7 Deployment.xml

```
1 <select id = "selectDeploymentsByName" resultMap = "deploymentResultMap"
2     parameterType = "org.activiti.engine.impl.db.ListQueryParameterObject" >
```



```

3      select * from ${prefix}ACT_RE_DEPLOYMENT D where NAME_ = # {parameter} order by D.
      DEPLOY_TIME_desc
4  </select>

```

上述 SQL 语句比较简单,以 NAME_ 字段为查询条件从 ACT_RE_DEPLOYMENT 表中查询数据,并按照流程文档的部署时间进行降序排列。

(2) 第 26 行如果 existingDeployments 集合不为空,则第 27 行需要取出该集合中的第一个元素,并使用变量 existingDeployment 进行存储。

(3) 第 30 行如果 existingDeployment 对象不为空,并且 deploymentsDiffer 方法的返回结果为 false(即将部署的流程文档已经存在于数据库中),则第 31 行直接将 existingDeployment 返回。deploymentsDiffer 方法用于对比 deployment 对象与 existingDeployment 对象是否相等,该方法的处理逻辑比较简单,由于篇幅有限,本书不做过多讲解。

7.2.2 设置标识位

deployment.setNew(true)操作设置了 deployment 对象中的 isNew 属性值为 true,该值非常重要,决定了该流程文档是否可以部署操作。既然 isNew 属性值可以设置为 true,那什么场景下设置为 false 呢? 下面结合图 7-3 来说明。

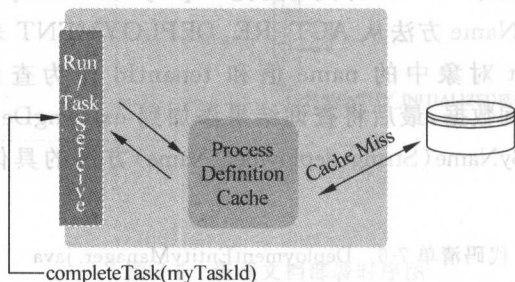


图 7-3 isNew 值的含义

看完图 7-3 的描述,对于 isNew 属性值设置为 false 的目的可能还是一知半解,为了便于理解,举个例子说明,例如现在要完成一个指定的任务,那么当开发人员调用 Activiti 提供的 API 完成任务时,引擎内部需要根据任务 id 值查找当前任务节点的信息,以及该任务节点的活动行为类、目标节点等信息,而以上所说的信息值均存储在流程定义缓存中,如果流程定义缓存数据丢失,则需要根据任务节点 id 值从数据库中查询该任务节点所归属的流程文档(存储于 ACT_GE_BYTEARRAY 表)以及流程定义信息(存储于 ACT_RE_PROCDEF 表),如果查询到流程文档信息,则会再次调用 BpmnDeployer 类中的 deploy 方法操作该流程文档,并设置 isNew 属性值设置为 false,该操作的目的是为了重新生成流程定义缓存数据,既然部署流程文档与重新生成流程定义缓存数据都需要调用 BpmnDeployer 类中的 deploy 方法,Activiti 就需要对这两个操作加以区分,这时 isNew 属性为 false 值就派上用场了,明确告诉 BpmnDeployer,现在只进行重新生成流程定义缓存数据的操作,不需要执行流程文档的入库操作。该过程至关重要但也非常复杂,稍后会更加深入讲解。

7.2.3 添加会话缓存

`commandContext.getDeploymentEntityManager().insertDeployment(deployment)` 操作主要是将 `deployment` 对象添加到会话缓存,引擎为何这样设计呢,为什么不直接将数据插入到数据库中呢?其实也不难理解,上文讲解了部署器分为三种:前置部署器、内置部署器、后置部署器。如果直接将数据添加到数据库中,就有可能衍生一个问题,既然数据都已经插入到数据库了,那么后置部署器已经没有存在的价值了,如果使用会话缓存技术,则会话缓存中的数据在彻底刷新到数据库之前,前置部署器可以对 `deployment` 对象的属性值进行操作,后置部署器可以对会话缓存中的数据进行任意修改,例如开发人员可以通过后置部署器获取会话缓存中的 `deployment` 对象并对其进行修改,也可以通过后置部署器将扩展元素信息值保存到指定的介质中,例如数据库或者内存。该过程可以使用图 7-4 进行通俗易懂的描述。关于会话缓存技术可以参考第 15 章。

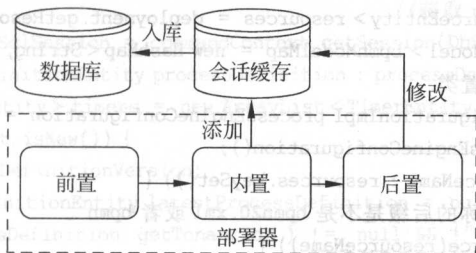


图 7-4 不同类型部署器的职责

7.2.4 部署管理器

`commandContext.getProcessEngineConfiguration().getDeploymentManager().deploy(deployment, deploymentSettings)` 开始部署流程文档,其处理逻辑可以分为两个步骤:获取部署管理器 `DeploymentManager`;调用 `DeploymentManager` 类的 `deploy` 方法部署流程文档,该方法的具体实现如代码清单 7-8 所示。

代码清单 7-8 `DeploymentManager.java`

```
1 public void deploy(DeploymentEntity deployment, Map<String, Object> deploymentSettings) {
2     for (Deployer deployer: deployers) {
3         deployer.deploy(deployment, deploymentSettings);
4     }
5 }
```

因为部署管理器中持有部署器集合, `deploy` 方法首先循环遍历 `deployers` 集合,然后调用 `deployer` 中的 `deploy` 方法部署流程文档。看到该方法的调用处理逻辑,就可以明白为何部署器集合使用 `List` 数据结构存储,因为这里是按照部署器集合的先后顺序进行遍历调用,并且 `deploy` 方法中的两个输入参数均为引用类型,这样开发人员就可以通过前置部署器或者后置部署器获取以上所说的两个输入参数,并进行相应的操作。

7.3 Bpmn 部署器

Activiti 将流程文档的部署行为抽象为 Deployer 接口,该接口在 Activiti 框架中占有非常重要的地位,Activiti 自身提供了两种实现类,即 BpmnDeployer 和 RulesDeployer(规则部署器)。本章节以 BpmnDeployer 类为例,详细讲解流程部署的整个过程,BpmnDeployer 类的相关定义如代码清单 7-9 所示。

代码清单 7-9 BpmnDeployer.java

```

1  public void deploy(DeploymentEntity deployment, Map<String, Object> deploymentSettings) {
2      //实例化存储流程定义实体的集合
3      List<ProcessDefinitionEntity> processDefinitions = new ArrayList<ProcessDefinition-
4          Entity>();
5      //根据部署对象获取部署实体
6      Map<String, ResourceEntity> resources = deployment.getResources();
7      Map<String, BpmnModel> bpmnModelMap = new HashMap<String, BpmnModel>();
8      //获取流程引擎配置类
9      ProcessEngineConfigurationImpl processEngineConfiguration =
10         Context.getProcessEngineConfiguration();
11     for (String resourceName : resources.keySet()) {
12         //验证流程资源名称的后缀是不是 bpmn20.xml 或者 bpmn
13         if (isBpmnResource(resourceName)) {
14             //根据 resourceName 获取 resource 对象
15             ResourceEntity resource = resources.get(resourceName);
16             byte[] bytes = resource.getBytes();
17             ByteArrayInputStream inputStream = new ByteArrayInputStream(bytes);
18             //实例化 BpmnParse 类,并为实例填充属性值
19             BpmnParse bpmnParse =
20                 bpmnParser.createParse().sourceInputStream(inputStream).setSourceSystemId(resourceName)
21                 .deployment(deployment).name(resourceName);
22             if (deploymentSettings != null) { //验证 deploymentSettings 参数值是否为空
23                 ...// 省略 Schema 验证
24             } else { ...//省略属性设置
25             }
26             bpmnParse.execute();
27             //循环遍历 bpmnParse 对象中的 processDefinitions 集合
28             for (ProcessDefinitionEntity processDefinition: bpmnParse.getProcessDefinitions()) {
29                 processDefinition.setResourceName(resourceName); //设置资源名称
30                 if (deployment.getTenantId() != null) {
31                     processDefinition.setTenantId(deployment.getTenantId()); //设置租户标识
32                 }
33                 String diagramResourceName = getDiagramResourceForProcess(resourceName,
34                     processDefinition.getKey(),resources); //获取流程文档的图片名称
35                 if (deployment.isNew()) { //判断是否流程文档需要重新部署
36                     ...//省略生成图片的代码
37                 }
38                 processDefinition.setDiagramResourceName(diagramResourceName);
39                 //设置 diagramResourceName 属性值

```

```

38 processDefinitions.add(processDefinition);
39 bpmnModelMap.put(processDefinition.getKey(), bpmnParse.getBpmnModel());
40 }
41 }
42 }
43 List<String> keyList = new ArrayList<String>();
44 for (ProcessDefinitionEntity processDefinition : processDefinitions) {
45     //遍历 processDefinitions 集合
46     if (keyList.contains(processDefinition.getKey())) {
47         throw new ActivitiException("same key processDefinition.getKey()");
48     }
49     keyList.add(processDefinition.getKey()); //将 processDefinition 对象的 key 值添加到 keyList 集合中
50 }
51 CommandContext commandContext = Context.getCommandContext();
52 ProcessDefinitionEntityManager processDefinitionManager =
53     commandContext.getProcessDefinitionEntityManager(); //获取 processDefinitionManager 对象
54 DbSqlSession dbSqlSession = commandContext.getSession(DbSqlSession.class);
55 for (ProcessDefinitionEntity processDefinition : processDefinitions) {
56     List<TimerEntity> timers = new ArrayList<TimerEntity>();
57     if (deployment.isNew()) {
58         //该变量存储流程定义版本号
59         int processDefinitionVersion;
60         ProcessDefinitionEntity latestProcessDefinition = null;
61         if (processDefinition.getTenantId() != null && !"".equals(processDefinition.getTenantId())) {
62             latestProcessDefinition =
63                 processDefinitionManager.findLatestProcessDefinitionByKeyAndTenantId(
64                     processDefinition.getKey(), processDefinition.getTenantId());
65         } else {
66             latestProcessDefinition = processDefinitionManager
67                 .findLatestProcessDefinitionByKey(processDefinition.getKey());
68         }
69         if (latestProcessDefinition != null) {
70             //如果该 key 的流程文档在数据库已经存在,则版本号加 1
71             processDefinitionVersion = latestProcessDefinition.getVersion() + 1;
72         } else {
73             processDefinitionVersion = 1; //版本从 1 开始计算
74         }
75         processDefinition.setVersion(processDefinitionVersion); //设置版本号
76         processDefinition.setDeploymentId(deployment.getId()); //设置部署 id 值
77         String nextId = idGenerator.getNextId(); //使用 id 生成器生成 id
78         String processDefinitionId = processDefinition.getKey() + ":" +
79             processDefinition.getVersion() + ":" + nextId;
80         if (processDefinitionId.length() > 64) {
81             //流程定义 id 值长度大于 64,需要进行值的截取
82             processDefinitionId = nextId;
83         }
84         processDefinition.setId(processDefinitionId);
85         //...省略转发 ENTITY_CREATED 事件
86         removeObsoleteTimers(processDefinition); //移除定时作业
87         addTimerDeclarations(processDefinition, timers); //添加定时作业

```



```

84     removeExistingMessageEventSubscriptions(processDefinition, latestProcessDefinition);
85     addMessageEventSubscriptions(processDefinition); //添加消息事件
86     removeExistingSignalEventSubscription(processDefinition, latestProcessDefinition);
87     addSignalEventSubscriptions(processDefinition); //添加信号事件
88     dbSqlSession.insert(processDefinition); //将 processDefinition 对象添加到会话缓存
89     addAuthorizations(processDefinition); //将流程启动人信息添加到会话缓存
90     //...省略转发 ENTITY_INITIALIZED 事件
91     scheduleTimers(timers); //调度定时作业
92 } else { //省略查询数据库的代码 }
93     DeploymentManager deploymentManager = processEngineConfiguration.getDeploymentManager();
94     //将 processDefinition 添加到缓存中
95     deploymentManager.getProcessDefinitionCache().add(processDefinition.getId(),
96     processDefinition);
97     //添加节点缓存
98     addDefinitionInfoToCache(processDefinition, processEngineConfiguration, commandContext);
99     deployment.addDeployedArtifact(processDefinition);
100    //操作节点缓存
101    createLocalizationValues(processDefinition.getId(),
102    bpmnModelMap.get ( processDefinition.getKey ( ) ). getProcessById ( processDefinition.
103    getKey ( ) ) );
104 }

```

根据代码量可以看出,部署器的处理逻辑相当复杂,围绕着流程文档的部署,引擎做了很多工作,该方法承载了太多的功能实现,为何不将该方法的处理逻辑分散到不同的方法中,这样看起来会更加的清晰明了,每一层的逻辑也会更加的单一、容易理解,这可能是 Activiti 需要优化的地方。

在细化讲解每一个细节实现之前,首先总览一下该方法,并将其进行如下总结。

(1) 首先初始化各种集合,第 3 行实例化 ArrayList 泛型类,该泛型类存储 ProcessDefinitionEntity 类型的元素,第 5 行从 deployment 对象中获取 resources 集合,该集合主要存储 ResourceEntity 类型的元素,第 6 行初始化 bpmnModelMap 集合,该集合存储 BpmnModel 类型的元素,第 8~9 行获取 ProcessEngineConfigurationImpl 实例对象。

(2) 获取并操作流程文档内容。

如果 isBpmnResource 方法 true,则程序执行第 13~39 行代码,如下所示:

- 第 18~20 行根据文件流 (inputStream)、资源名称 (resourceName)、部署对象 (deployment) 直接调用 bpmnParser 对象中的 createParse 方法创建 BpmnParse 实例对象,BpmnParse 实例对象非常重要,负责将流程文档解析之后的 BaseElement 实例对象再次进行解析并注入流程虚拟机。
- 第 21~24 行如果 deploymentSettings 参数值不为空,则可以根据该参数值设置是否开启流程文档验证功能,如果 deploymentSettings 参数值为空,则设置不需要进行流程文档的验证工作。该判断操作非常重要,例如流程定义缓存丢失,则程序再次对流程文档解析时,就没有必要对其进行验证了。
- 第 25 行 bpmnParse.execute() 方法负责将流程文档中的元素解析并封装为 Activiti 中的内部表示 BaseElement 实例,然后再次解析 BaseElement 实例对象并

将其注入流程虚拟机。

- bpmnParse.execute()方法执行完毕,开始循环遍历 bpmnParse 对象持有的 processDefinitions 集合并对 processDefinition 对象进行属性填充,填充的属性有资源名称 resourceName、租户标识 tenantId 等。如果 deployment.isNew()方法返回值为 true,则需要根据流程文档信息生成图片并将其添加到会话缓存中。如果流程文档部署之后,生成的图片有中文乱码的情况,可以查看生成图片的处理逻辑。

以上一系列的操作完毕,将相应的信息值添加到 processDefinitions 集合和 bpmnModelMap 集合中。processDefinitions 集合中是否有值决定了程序是否可以进行下一步的处理。

注意

Activiti 使用 Java 语言提供的 Graphics2D 类库绘制图片,关于图片的生成逻辑可以跟进 DefaultProcessDiagramCanvas 类进行查看。开发人员可以通过设置 processDiagramGenerator 开关属性注入自定义图片生成类。

(3) 保证流程定义 key 唯一性。

第 43 行初始化 keyList 集合,然后循环遍历 processDefinitions 集合,将 processDefinition 对象中的 key 添加到 keyList 集合中,添加时如果发现该 key 已经存在 keyList 集合中,程序直接报错。

(4) 获取流程定义实体管理类。

通过 Context 类获取到 CommandContext 实例,然后通过 commandContext 对象获取流程定义实体管理类 ProcessDefinitionEntityManager 实例对象(负责 ACT_RE_PROCDEF 表的操作)。

(5) DbSqlSession 对象获取。

通过 commandContext 对象获取到 DbSqlSession 实例对象,该实例对象负责会话缓存的添加、删除以及更新操作。

(6) 第 54 行循环遍历 processDefinitions,如果 deployment.isNew()方法的返回值为 true,则执行第 57~91 行的代码;否则执行第 92 行代码直接从数据库查询数据。其中第 57~91 行的代码执行逻辑如下:

- 第 58~66 行获取 ProcessDefinitionEntity 实例对象。
- 第 67~71 行计算流程定义版本值。
- 第 75~79 行计算流程定义 id 值。
- 第 82~83 行移除过期作业和添加新的作业。
- 第 84~85 行移除过期消息及添加新的消息。
- 第 86~87 行移除过期信号及添加新的信号。
- 第 88 行将 processDefinition 添加到会话缓存。
- 第 91 行调度定时作业。可以参考第 9 章。

(7) 第 93 行获取 DeploymentManager 实例对象。

(8) 第 95~96 行将 processDefinition 对象添加到缓存中。

(9) 第 98~99 行开始添加节点缓存。

(10) 第 101~102 行开始操作节点缓存。

接下来细化讲解该方法的具体实现逻辑。第 93~102 行关于缓存的相关操作可以参考第 8 章。

7.3.1 获取资源信息

deployment.getResources() 主要用于获取 resources 集合, 相关实现如代码清单 7-10 所示。

代码清单 7-10 DeploymentEntity.java

```
1 protected Map<String, ResourceEntity> resources;
2 public Map<String, ResourceEntity> getResources() {
3     if (resources == null && id != null) {
4         List<ResourceEntity> resourcesList = Context.getCommandContext().getResourceEntity-
5             yManager()
6             .findResourcesByDeploymentId(id);
7         resources = new HashMap<String, ResourceEntity>();
8         for (ResourceEntity resource: resourcesList) {
9             resources.put(resource.getName(), resource);
10        }
11    }
12    return resources;
13 }
```

在上述代码中, 使用懒加载方式获取 resources 集合, 具体实现逻辑如下。

(1) resources 属性值判断。

首先, 第 3 行判断当前对象中的 resources 集合是否为空, 如果该集合不为空或者 id 值为空, 则第 11 行直接返回该集合, 否则进行如下的处理。

(2) 从数据库获取。

第 4~5 行以 id 值(流程部署 id)为查询条件从数据库(ACT_GE_BYTEARRAY 表)中查找数据, 第 6 行初始化 resources 集合, 然后第 7~8 行循环遍历 resourcesList 集合并将该集合中的值添加到 resources 集合中。客户端部署流程文档的时候可以指定 resourceName 值, 形如 repositoryService.createDeployment().addInputStream(resourceName, inputStream)。

强调

resources 集合中 key 的值对应表 ACT_GE_BYTEARRAY 中的 NAME_字段值。

7.3.2 封装资源信息

上面讲解了 DeploymentEntity 类中 resources 集合的获取过程, 那么该集合何时初始化呢, 这就涉及了流程文档的部署操作, 部署流程文档的过程如代码清单 7-11 所示。

代码清单 7-11 App.java

```
1 InputStream inputStream = Pvm.class.getClassLoader().getResourceAsStream("pvm.bpmn20.xml");
2 String category = "pvm"; // 流程分类
```

```
3 String resourceName = "pvm.bpmn";
4 repositoryService.createDeployment().addInputStream(resourceName, inputStream).deploy();
```

上述代码中,第4行 addInputStream 方法的具体实现如代码清单 7-12 所示。

代码清单 7-12 DeploymentBuilderImpl.java

```
1 protected DeploymentEntity deployment = new DeploymentEntity();
2 public DeploymentBuilder addInputStream(String resourceName, InputStream inputStream) {
3     if (inputStream == null) {
4         throw new ActivitiIllegalArgumentException("inputStream for resource '" + resourceName + "'
is null");
5     }
6     byte[] bytes = IoUtil.readInputStream(inputStream, resourceName);
7     ResourceEntity resource = new ResourceEntity(); //实例化 ResourceEntity 类
8     resource.setName(resourceName);
9     resource.setBytes(bytes);
10    deployment.addResource(resource);
11    return this;
12 }
```

在上述代码中,addInputStream 方法的处理逻辑如下。

- (1) 第3行对 inputStream 参数进行非空校验,如果该参数为空,则程序报错。
- (2) 第6行读取 inputStream 流并将其转化为 byte 数组。
- (3) 第7行实例化 ResourceEntity 类,并对其进行属性填充。
- (4) 第10行将 resource 对象添加到 deployment 对象中,其中 addResource 方法的实现如代码清单 7-13 所示。

代码清单 7-13 DeploymentEntity.java

```
1 protected Map<String, ResourceEntity> resources;
2 public void addResource(ResourceEntity resource) {
3     if (resources == null) {
4         resources = new HashMap<String, ResourceEntity>();
5     }
6     resources.put(resource.getName(), resource);
7 }
```

在上述代码中,第3行判断 resources 集合是否为空,如果为空,则第4行初始化该集合,第6行将 resource 对象添加到 resources 集合中。

注意

resources 为 Map 数据结构, key 为资源名称, value 为 ResourceEntity 实例对象。

7.3.3 校验资源名称

上面讲解了 resources 集合的获取以及初始化逻辑,接下来分析遍历该集合的具体处理

流程,首先分析 isBpmnResource 方法的实现,如代码清单 7-14 所示。

代码清单 7-14 BpmnDeployer.java

```
1 public static String[] BPMN_RESOURCE_SUFFIXES = new String[] { "bpmn20.xml", "bpmn" };
2 protected boolean isBpmnResource(String resourceName) {
3     for (String suffix : BPMN_RESOURCE_SUFFIXES) {
4         if (resourceName.endsWith(suffix)) {
5             return true;
6         }
7     }
8     return false;
9 }
```

在上述代码中,第 3~6 行循环遍历 BPMN_RESOURCE_SUFFIXES 集合,然后判断 resourceName 参数值是否规范的 Bpmn 资源名称,如果该参数值的后缀是 bpmn20.xml 或者 bpmn 中的任意一个,则该方法返回 true。该方法的处理逻辑虽然很简单,但却至关重要,因为该方法的返回值直接决定了流程定义数据是否可以添加到流程定义表 ACT_RE_PROCDEF 中,如果该方法的返回值为 false,则引擎不会将流程定义数据添加到流程定义表中,该表中是否存在数据直接决定程序能否启动流程实例。

7.3.4 计算流程定义版本值

程序首先根据流程定义 key 值或者流程定义租户标识 tenantId 值作为查询条件,从 ACT_RE_PROCDEF 表中查询数据,如果 tenantId 不存在或者为空,则委托 ProcessDefinitionManager 类中的 findLatestProcessDefinitionByKey 方法查询数据;否则委托 ProcessDefinitionManager 类中的 findLatestProcessDefinitionByKeyAndTenantId 方法查询数据,不管程序调用哪个方法查询数据,最终均是通过 MyBatis 框架查询 ACT_RE_PROCDEF 表。

查询操作执行完毕开始计算流程定义的版本值,例如 ACT_RE_PROCDEF 表中存在 key 值为“shareniu”并且版本为 2 的流程定义数据,则当前的流程定义版本为 3(加 1 操作);如果 ACT_RE_PROCDEF 表中不存在 key 值为“shareniu”的数据则流程定义版本值默认从 1 开始。

7.3.5 生成流程定义 id 值

这里所说的流程定义 id 对应 ACT_RE_PROCDEF 表的主键 ID_值,id 值生成策略为流程定义 key+":"+流程定义版本+":"+id 生成器生成的 id 值,如果生成的 id 值长度大于 64,则直接使用 id 生成器生成的值作为 id 值。

为什么 id 值长度必须小于 64 位呢?在这里简单说明一下,因 ACT_RE_PROCDEF 表中的 ID_列是 varchar 类型并且最大长度为 64,所以程序必须限制生成的 id 值长度,否则长度超出限制,数据插入数据库时程序报错。能否使其长度不受限制呢?当然可以,其实现思路为:①需要考虑如何修改 deploy 方法中 id 值的生成逻辑;②需要修改 ACT_RE_

PROCDEF 中的 ID_列的长度,但只修改该表中的 ID_列长度还是远远不够的,该表中的 ID_列作为其他表外键约束列的地方也要进行修改。

7.3.6 移除过期作业

removeObsoleteTimers 方法用于移除过期作业,该操作主要针对配置了定时作业的流程。关于定时作业的配置可以参考第 9 章。接下来分析 removeObsoleteTimers 方法的具体实现,如代码清单 7-15 所示。

代码清单 7-15 BpmnDeployer.java

```

1 protected void removeObsoleteTimers(ProcessDefinitionEntity processDefinition) {
2     List<Job> jobsToDelete = null;
3     if (processDefinition.getTenantId() != null && !"".equals(processDefinition.
4         getTenantId())){
5         jobsToDelete = Context.getCommandContext().getJobEntityManager().
6             findJobsByTypeAndProcessDefinitionKeyAndTenantId(
7                 TimerStartEventJobHandler.TYPE, processDefinition.getKey(),
8                 processDefinition.getTenantId());
9     } else {
10        jobsToDelete = Context.getCommandContext().getJobEntityManager().
11            findJobsByTypeAndProcessDefinitionKeyNoTenantId(
12                TimerStartEventJobHandler.TYPE, processDefinition.getKey());
13    }
14    if (jobsToDelete != null) {
15        for (Job job : jobsToDelete) {
16            new CancelJobsCmd(job.getId()).execute(Context.getCommandContext());
17        }
18    }
19 }

```

在上述代码中,第 3 行判断 processDefinition.getTenantId() 值是否为空,并以 tenantId 值是否存在为分水岭执行不同的逻辑,如果 tenantId 存在则执行第 4~7 行代码委托 JobEntityManager 类中的 findJobsByTypeAndProcessDefinitionKeyAndTenantId 方法查找数据;如果 tenantId 不存在,则执行第 9~11 行代码委托 JobEntityManager 类中的 findJobsByTypeAndProcessDefinitionKeyNoTenantId 方法查找数据,以上操作就是根据不同的条件检索数据库 ACT_RU_JOB 表中的数据,并使用 jobsToDelete 集合存储查询之后的数据。

代码第 13~16 行判断 jobsToDelete 集合是否为空,如果不为空,则循环遍历该集合并调用 CancelJobsCmd 命令类移除作业(删除 ACT_RU_JOB 表中查询的数据)。

看到这里的处理逻辑,可能会有疑问:开发人员只是单纯的部署流程文档,为何需要移除定时作业?试想一下,如果开发人员部署了一个 key 为 "shareniu" 的流程文档,并指定该流程一天之后启动,则该流程文档部署完毕版本为 1,如果在该流程等待启动时间内,开发人员再次部署该流程文档并重新设置启动时间为两天,则该流程文档部署之后版本为 2,这时会产生一个问题,因为流程定义 key 相同且多个版本共存的情况下,流程实例默认是按照

最高版本启动的,这时第一个版本的流程实例是否需要在指定时间启动已经意义不大,只需要让第二个版本的流程文档在指定时间启动即可,对于该场景来说,第一个版本的定时作业需要通过 `removeObsoleteTimers` 方法删除,并通过 `addTimerDeclarations` 方法重新添加第二个版本的定时作业,`addTimerDeclarations` 方法操作的数据库表为 `ACT_RU_JOB`。下面分析 `addTimerDeclarations` 方法的实现。

7.3.7 添加作业

`addTimerDeclarations` 方法的具体实现如代码清单 7-16 所示。

代码清单 7-16 BpmnDeployer.java

```
1 protected void addTimerDeclarations(ProcessDefinitionEntity processDefinition,  
2 List<TimerEntity> timers) {  
3     List<TimerDeclarationImpl> timerDeclarations = (List<TimerDeclarationImpl>)  
4     processDefinition.getProperty(BpmnParse.PROPERTYNAME_START_TIMER);  
5     if (timerDeclarations != null) {  
6         for (TimerDeclarationImpl timerDeclaration : timerDeclarations) {  
7             TimerEntity timer = timerDeclaration.prepareTimerEntity(null);  
8             if (timer != null) {  
9                 timer.setProcessDefinitionId(processDefinition.getId());  
10                if (processDefinition.getTenantId() != null) {  
11                    timer.setTenantId(processDefinition.getTenantId());  
12                }  
13                timers.add(timer);  
14            }  
15        }  
16    }  
17 }
```

在上述代码中,第 3~4 行获取 `timerDeclarations` 集合,如果该集合为空,则不予处理,否则开始执行第 6~13 行代码,其中第 7 行构造 `timer` 对象,如果该对象不为空,则执行第 9~11 行对其进行属性填充,第 13 行将其添加到 `timers` 集合中,如果 `timers` 存在值,则 `scheduleTimers(timers)` 方法会对该集合进行处理。

7.3.8 处理消息

不管是消息启动节点或者信号启动节点,处理流程都是首先移除已经存在的或者过期的数据,然后重新添加新的数据,该操作针对的是 `ACT_RU_EVENT_SUBSCR` 表,信号启动与消息启动方式在该表中唯一区别就是 `EVENT_TYPE` 列类型不同,消息启动节点类型为 `message`,信号启动类型为 `signal`。消息启动与信号启动的删除和添加逻辑非常简单,不再详细介绍,可以参考作业的移除和添加讲解。下面讲解消息启动的使用,首先分析消息启动节点的配置,如代码清单 7-17 所示。

代码清单 7-17 paymentMessage.bpmn

```
1 <!-- 定义一个消息元素 -->
2 <message id="paymentMessage" name="paymentMessage"></message>
3 <process id="operationProcess" name="process" isExecutable="true">
4   <userTask id="usertask1" name="User Task"></userTask>
5   <startEvent id="messagestartevent1" name="Message start">
6     <!-- 消息启动节点 messageRef 对应 id 为 pay 的消息 -->
7     <messageEventDefinition id="msg" messageRef="pay"></messageEventDefinition>
8   </startEvent>
9 </process>
```

在上述代码中,定义了附有消息启动节点的流程文档,该流程文档部署之后,启动方式如代码清单 7-18 所示。

代码清单 7-18 App.java

```
1 public void startProcessInstanceByMessage() {
2   runtimeService.startProcessInstanceByMessage("paymentMessage");
3 }
```

在上述代码中,第 2 行中的 "paymentMessage" 值对应代码清单 7-17 中第 2 行定义的 id 值。

7.3.9 处理信号

信号启动节点的配置如代码清单 7-19 所示。

代码清单 7-19 signalEvent.bpmn

```
1 <!-- 定义一个信号元素 -->
2 <signal id="signalEvent" name="signalEvent"></signal>
3 <process id="signalProcess" name="process" isExecutable="true">
4   <startEvent id="signalstartevent1" name="Signal start">
5     <!-- 信号启动 signalRef 对应 signal 元素中的 id 值 -->
6     <signalEventDefinition id="sig" signalRef="signalEvent"></signalEventDefinition>
7   </startEvent>
8 </process>
```

在上述代码中,定义了附有信号启动节点的流程文档,该流程文档部署之后,启动的方式如代码清单 7-20 所示。

代码清单 7-20 App.java

```
1 public void startProcessInstanceByMessage() {
2   runtimeService.signalEventReceived("signalEvent");
3 }
```


在上述代码中,第 2 行中的"signalEvent"值对应代码清单 7-19 中第 2 行定义的 id 值。

7.3.10 设置流程启动人

addAuthorizations(processDefinition)方法负责将流程文档中定义的流程启动人添加到会话缓存中,该方法的具体实现如代码清单 7-21 所示。

代码清单 7-21 BpmnDeployer.java

```

1  protected void addAuthorizations(ProcessDefinitionEntity processDefinition) {
2      addAuthorizationsFromIterator(processDefinition.getCandidateStarterUserIdExpressions(),
3      processDefinition, ExprType.USER);                                //添加用户
4      addAuthorizationsFromIterator(processDefinition.getCandidateStarterGroupIdExpressions(),
5      processDefinition, ExprType.GROUP);                                //添加组
6  }
7  void addAuthorizationsFromIterator(Set<Expression> exprSet, ProcessDefinitionEntity processDefinition, ExprType exprType) {
8      if (exprSet != null) {                                            //判断 exprSet 参数是否为空
9          Iterator<Expression> iterator = exprSet.iterator();          //循环遍历 exprSet 集合
10         while (iterator.hasNext()) {
11             Expression expr = (Expression) iterator.next();
12             IdentityLinkEntity identityLink = new IdentityLinkEntity();
13             identityLink.setProcessDef(processDefinition);
14             if (exprType.equals(ExprType.USER)) {
15                 identityLink.setUserId(expr.toString());              //设置用户
16             } else if (exprType.equals(ExprType.GROUP)) {
17                 identityLink.setGroupId(expr.toString());              //设置组
18             }
19             identityLink.setType(IdentityLinkType.CANDIDATE);          //设置用户的类型
20             identityLink.insert();                                       //添加到缓存
21         }
22     }
23 }

```

在上述代码中,第 2~3 行用于添加用户,第 4~5 行用于添加组,对第 7 行定义的 addAuthorizationsFromIterator 方法的处理逻辑梳理总结为:第 8 行对 exprSet 参数值进行非空校验,如果该参数值为空,则程序不予处理;如果该参数值不为空,则首先遍历 exprSet 集合。第 12 行实例化 IdentityLinkEntity 类,然后为 identityLink 对象填充如下属性:流程定义、类型、处理人、处理组,最后第 20 行调用 identityLink.insert()方法将其添加到会话缓存,该步骤操作的表为 ACT_RU_IDENTITYLINK。接下来分析 identityLink.insert()方法的实现逻辑,如代码清单 7-22 所示。

代码清单 7-22 IdentityLinkEntity.java

```

1  public void insert() {
2      Context.getCommandContext().getDbSqlSession().insert(this);
3      Context.getCommandContext().getHistoryManager().recordIdentityLinkCreated(this);
4  }

```

insert 方法的处理非常精巧,首先第 2 行将自身(this)添加到会话缓存中,然后第 3 行将自身(this)添加到历史权限表中了,该操作可谓一箭双雕,部署流程文档的同时,不仅完成了 ACT_RU_IDENTITYLINK 表的数据插入,还完成了 ACT_HI_IDENTITYLINK 表数据的插入。只有当客户端需要操作 Activiti 时,才会和 Activiti 建立一个短连接,操作执行完毕,连接立刻被关闭,相应的资源也会被回收。

7.4 自定义部署器实战

上文详细讲解了 BpmnDeployer 类中 deploy 方法的处理逻辑,其中使用 BpmnDeployer 类进行流程文档部署操作时,首先会验证资源名称的后缀是不是以 bpmn20.xml 或者 bpmn 结尾,如果不是,则无法部署流程文档,最终导致程序无法启动流程实例,这时就需要考虑如何扩展 BpmnDeployer 类,幸运的是 Activiti 为开发人员提供了一系列的开关属性,以方便客户端扩展,本案例详细讲解 BpmnDeployer 类的扩展,首先定义一个类并继承 BpmnDeployer 类,相关定义如代码清单 7-23 所示。

代码清单 7-23 ShareniuBpmnDeployer.java

```
1 public class ShareniuBpmnDeployer extends BpmnDeployer {
2     public static final String[] BPMN_RESOURCE_SUFFIXES = new String[] { "bpmn20.xml",
3     "bpmn", "shareniu" };
4     protected boolean isBpmnResource(String resourceName) {
5         for (String suffix : BPMN_RESOURCE_SUFFIXES) {
6             if (resourceName.endsWith(suffix)) {
7                 return true;
8             }
9         }
10        return false;
11    }
12 }
```

ShareniuBpmnDeployer 类的定义非常简单,直接覆盖 BpmnDeployer 类中的 isBpmnResource 方法,并在该方法中检验流程资源名称的后缀。接下来的工作就是将 ShareniuBpmnDeployer 注入流程引擎配置类,具体实现如代码清单 7-24 所示。

代码清单 7-24 activiti.cfg.xml

```
1 <bean id="processEngineConfiguration"
2     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <property name="bpmnDeployer" ref="shareniuBpmnDeployer">
4     </property>
5 </bean>
6 <!-- 配置自定义部署器 -->
7 <bean id="myBpmnDeployer" class="com.shareniu.chapter7.ShareniuBpmnDeployer"></bean>
```

可以按照该案例的扩展思路,定义一个流程文档并对其进行部署,进而验证 ShareniuBpmnDeployer 类是否生效。

来源: 中国书画函授大学肇庆分校建校二十周年纪念册

8.1 背景

(1) 使 Activiti 的内存占用率低, 因为只有需要获取流程实例或者任务信息等操作时才会查询数据库。

(2) 允许在一个集群环境下(共享数据库)运行 Activiti, 集群中的节点相互之间不需要通信, 因为数据库才是唯一真实的数据来源, 当集群中的任意节点需要获取一些流程状态时, 可以直接从数据库取出或者写入。在 Activiti 引擎执行操作期间, MyBatis 框架(底层使用的 ORM 框架)会保持一个“会话缓存”来避免同一个会话多次从数据库中获取相同的数据, 例如主键值, 然而这个“会话缓存”的生命周期非常短暂, 可以用朝生夕死来形容, 因此“会话缓存”并不会危及集群运行 Activiti 的能力。关于会话缓存的构造以及使用过程可以参考第 15 章。

上面提到了 Activiti 工作方式是无状态的,因为通常情况下 Activiti 引擎运行在 Web 容器,例如 Tomcat、Weblogic 等,并通过 HTTP 方式提供服务,由于 HTTP 协议是无状态的,因此可以说 Activiti 工作方式也是无状态的,当然,Activiti 自身也不会记录操作者以及操作的数据,只有当客户端需要操作 Activiti 时,才会和 Activiti 建立一个短连接,操作执行完毕,连接立刻被关闭,相应的资源也会被回收。

由于缓存所有的流程数据可能会占用系统资源和内存,所以通常情况下开发人员期望将永远不会改变的数据作为优先考虑的缓存对象,例如流程文档。开发人员提供流程文档,流程引擎解析流程文档的元素并将其转化为可以执行的结构(更具体地说,就是将流程文档解析转化为 Activiti 内部 POJO 树),解析工作完成之后,将流程文档的内容以及必要信息存储到数据库,例如描述信息、业务键等,以上过程产生的流程文档永远不会改变(除非人为进行修改),直到流程被删除,该过程非常消耗系统资源(XML 解析总是如此),因此 Activiti 使用缓存来存储流程文档解析之后的结果。这样当启动流程实例或者完成任务时首先尝试从缓存中加载,如果缓存中没有值,则开始查询数据库并再次执行流程文档的解析,最终将解析结果存储到缓存中。

8.2 初始化缓存策略

在第 7 章中详细讲解了流程文档部署的整个过程,其中涉及了缓存处理类的实例化操作,下面以缓存处理类为切入点,深入探究 Activiti 中缓存的应用场景,如代码清单 8-1 所示。

代码清单 8-1 ProcessEngineConfigurationImpl.java

```

1  protected void initDeployers() {
2      if (deploymentManager == null) {          //部署管理器
3          deploymentManager = new DeploymentManager();
4          deploymentManager.setDeployers(deployers);
5          if (processDefinitionCache == null) {
6              if (processDefinitionCacheLimit <= 0) {
7                  processDefinitionCache = new DefaultDeploymentCache<ProcessDefinitionEntity>();
8              } else {
9                  processDefinitionCache = new
10                     DefaultDeploymentCache<ProcessDefinitionEntity>(processDefinitionCacheLimit);
11              }}
12          if (bpmnModelCache == null) {
13              if (bpmnModelCacheLimit <= 0) {
14                  bpmnModelCache = new DefaultDeploymentCache<BpmnModel>();
15              } else {
16                  bpmnModelCache = new DefaultDeploymentCache<BpmnModel>(bpmnModelCacheLimit);
17              }}
18          if (processDefinitionInfoCache == null) {
19              if (processDefinitionInfoCacheLimit <= 0) {
20                  processDefinitionInfoCache = new ProcessDefinitionInfoCache(commandExecutor);
21              } else {
22                  processDefinitionInfoCache = new ProcessDefinitionInfoCache(commandExecutor,

```



```

23     processDefinitionInfoCacheLimit);
24 }
25 if (knowledgeBaseCache == null) {
26     if (knowledgeBaseCacheLimit <= 0) {
27         knowledgeBaseCache = new DefaultDeploymentCache<Object>();
28     } else {
29         knowledgeBaseCache = new DefaultDeploymentCache<Object>(knowledgeBaseCacheLimit);
30     }
31 deploymentManager.setProcessDefinitionCache(processDefinitionCache);
32 deploymentManager.setBpmnModelCache(bpmnModelCache);
33 deploymentManager.setProcessDefinitionInfoCache(processDefinitionInfoCache);
34 deploymentManager.setKnowledgeBaseCache(knowledgeBaseCache);
35 }
36 }

```

虽然在上述代码中四种缓存处理类的职责不同,但是处理逻辑大体相同,将其总结如下。

(1) 客户端自定义缓存处理类判断。

以上四种缓存处理类的处理逻辑均是首先判断客户端是否自定义了缓存处理类(开关属性),如果客户端自定义了缓存处理类,则直接使用,否则使用系统内置的缓存处理类,在实例化系统内置的缓存处理类时需要根据缓存对象的容器大小限制值进行判断。

(2) 系统内置缓存处理类策略判断。

首先获取客户端配置的缓存对象的容器大小限制值(开关属性),以上四种缓存限制值默认为-1,也就是说默认对缓存对象的容器大小不进行限制,其内部使用 Map 进行实现,当限制缓存容器大小时,则使用 LRU 进行算法控制。程序以缓存限制值为分水岭执行不同的逻辑,不管缓存限制值的大小为多少,都会根据缓存限制值实例化不同的缓存处理类,第 6~10 行实例化 DefaultDeploymentCache 泛型类,该泛型类存储 ProcessDefinitionEntity 实例对象,第 13~16 行实例化 DefaultDeploymentCache 泛型类,该泛型类存储 BpmnModel 实例对象,第 19~23 行实例化 ProcessDefinitionInfoCache 类,该类负责管理节点缓存,第 27~29 行实例化 DefaultDeploymentCache 泛型类,该泛型类存储 KnowledgeBase 实例对象。DefaultDeploymentCache 类为默认的缓存处理类。

(3) 第 31~34 行将以上所述的四个缓存处理类设置到 DeploymentManager 实例对象中。

8.3 部署管理器

上文提到了 DeploymentManager 类,该类的管理范围如图 8-1 所示。

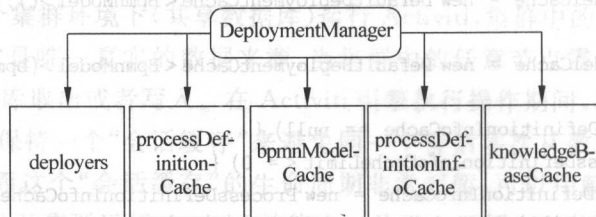


图 8-1 DeploymentManager 类的管控范围

DeploymentManager 类的核心定义如代码清单 8-2 所示。

代码清单 8-2 DeploymentManager.java

```
1 DeploymentCache<ProcessDefinitionEntity> processDefinitionCache;  
2 DeploymentCache<BpmnModel> bpmnModelCache;  
3 ProcessDefinitionInfoCache processDefinitionInfoCache;  
4 DeploymentCache<Object> knowledgeBaseCache;
```

在上述代码中,第 1、2、4 行中使用的类为 DeploymentCache,第 3 行使用的类为 ProcessDefinitionInfoCache。

8.4 缓存处理类架构

默认的缓存处理类为 DefaultDeploymentCache,该类实现了 DeploymentCache 接口,DeploymentCache 接口的类图如图 8-2 所示。

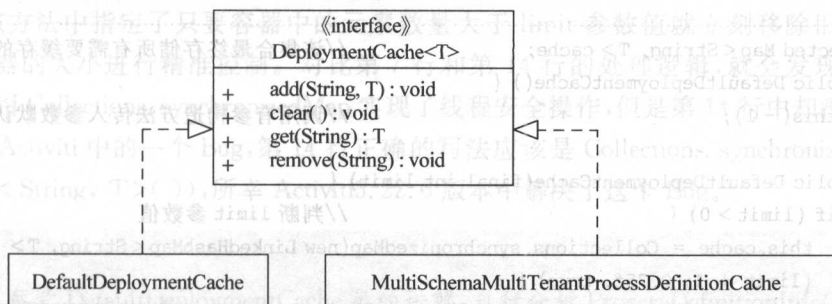


图 8-2 DeploymentCache 类图

根据图 8-2 可以很清晰地看到 DeploymentCache 接口的定义及其实现类,图中类的职责大致如下。

- (1) DeploymentCache: 该接口定义了缓存数据的添加、获取、移除以及清空四个方法。
- (2) DefaultDeploymentCache: 该类对接口 DeploymentCache 中的方法进行了实现,其内部使用 Map 数据结构维护所有的缓存数据,包括使用 HashMap 以及 LinkedHashMap (LRU 算法实现的基石)。
- (3) MultiSchemaMultiTenantProcessDefinitionCache: “多数据库多租户”流程定义缓存。

DeploymentCache 接口中并没有定义更新缓存的方法,原因很简单,因为该接口的默认实现类内部均使用 Map 数据结构来维护缓存中的数据,基于 Map 数据结构的特性,如果用户期望更新数据,只需要调用 add 方法即可完成对旧数据的更新,因此没有必要在该接口中单独定义更新方法,但是这样设计就会衍生一个问题,如果客户端不打算使用 DefaultDeploymentCache 作为缓存处理类,那么该如何实现更新缓存的功能呢? 这里提供如下两种实现思路。

- (1) 定义一个接口并继承 DeploymentCache 接口,在该接口中定义一个更新缓存的方

法供客户端使用,这样实现起来虽然有点烦琐,但可以保证每个方法的功能更加单一,也方便后期维护。

(2) 自定义一个缓存处理类并实现 `DeploymentCache` 接口,在 `add` 方法中实现缓存的添加以及更新逻辑。

可能有些开发人员喜欢使用 `clear` 方法清除缓存数据,使用 `clear` 方法清除缓存数据归根结底是对 Activiti 中的缓存处理机制不了解所导致的误操作,因为该操作会清除缓存中所有的数据,如果项目中流程文档比较多,缓存对象势必会非常多,如果贸然调用 `clear` 方法清除所有的缓存数据,极易导致缓存崩溃,程序性能急剧下降,因此不推荐使用 `clear` 方法清除缓存数据,最好使用 `remove` 方法移除指定的缓存数据。

8.5 默认缓存处理类及 Bug

`DefaultDeploymentCache` 类的核心定义如代码清单 8-3 所示。

代码清单 8-3 `DefaultDeploymentCache.java`

```
1  protected Map<String, T> cache; //该集合最终存储所有需要缓存的对象
2  public DefaultDeploymentCache() {
3      this(-1); //调用有参构造方法传入参数默认值为-1
4  }
5  public DefaultDeploymentCache(final int limit) {
6      if (limit > 0) { //判断 limit 参数值
7          this.cache = Collections.synchronizedMap(new LinkedHashMap<String, T>
8              (limit + 1, 0.75f, true));
9          protected boolean removeEldestEntry(Map.Entry<String, T> eldest) {
10             boolean removeEldest = size() > limit; //判断是否可以移除集合中的数据
11             return removeEldest;
12         }
13     };
14 } else {this.cache = new HashMap<String, T>(); //初始化 cache 集合}
15 }
```

将上面代码的执行逻辑梳理总结如下。

(1) 实例化缓存处理类。

`DefaultDeploymentCache` 类为泛型类,如果客户端实例化该类的同时指定了泛型类型,则该类的成员变量 `cache` 类型已经确定,并且该类第 2 行定义的无参构造方法会调用第 5 行定义的构造方法。

(2) Map 缓存策略选用。

在第 5 行定义的构造方法中,如果 `limit` 参数值小于 0 或者等于 0,则表示缓存对象的容器大小不受限制,直接调用第 14 行代码初始化 `cache` 集合并将其作为缓存对象的容器。

在此说明实际项目开发中该操作是容易出错的地方:如果开发的工作流平台系统是同时让很多外部系统访问使用的,且每个系统使用的数据库相互之间独立运行,因此不同系统下的流程文档进行部署时可能生成重复的流程定义 `id` 值(`ACT_RE_PROCDEF` 的 `ID_` 字段),

例如 A 系统和 B 系统都存在名为 shareniu:1:1 的流程,由于 Activiti 使用 Map 数据结构存储缓存对象,如果 key 相同则只会加载其中的一个对象到内存中,哪个系统先使用则优先加载,因此就可能出现 A 系统使用 B 系统流程的问题,或者 B 系统使用 A 系统流程的问题。

解决方案:可以在流程文档部署时,添加租户标识 tenantId 对系统的来源进行区分,也可以在流程文档定义阶段为流程定义的 key 值添加系统标识,从源头上解决问题,这样就可以保证在全系统中流程定义 id 值的唯一性。

(3) LinkedHashMap 缓存。

在第 6~13 行中,如果 limit 参数值大于 0,则需要限制 cache 容器的大小为 limit 参数值,关于缓存的 LRU 算法,Activiti 使用 LinkedHashMap 进行实现,因为 LinkedHashMap 默认就是按照元素的添加顺序进行存储,当然了也支持按照元素的访问顺序进行存储,即最新访问的数据在容器的最前面,LinkedHashMap 类中定义了判断是否需要移除最老数据的方法 removeEldestEntry,该方法默认返回值为 false,即不移除数据。

在上述代码中,第 7 行实例化 LinkedHashMap 类,并使用 Collections.synchronizedMap 方法实现线程安全操作,并在实例化 LinkedHashMap 的同时设置第三个输入参数值为 true,即按照访问顺序对元素进行排序。第 9~12 行重写了 LinkedHashMap 中的 removeEldestEntry 方法,并在该方法中指定了只要容器中的元素数量大于 limit 参数值就立刻移除旧数据,从而可以对容器的大小进行精准控制。对比第 7 行和第 14 行的处理逻辑,就会发现一个问题,第 7 行使用 Collections.synchronizedMap 实现了线程安全操作,但是第 14 行中却没有使用,很显然这是 Activiti 中的一个 Bug,第 14 行正确的写法应该是 Collections.synchronizedMap(new HashMap<String, T>()),所幸 Activiti 5.22.0 版本中解决了这个 Bug。

建议

可以参考 DefaultDeploymentCache 类的讲解,自行分析 ProcessDefinitionInfoCache 类。

8.6 流程定义缓存

上面详细分析了 DefaultDeploymentCache 类的架构及其处理逻辑,因为该类内部使用 Map 集合缓存数据,对于客户端而言,缓存数据的获取、添加以及更新操作不方便而且不灵活,例如系统重启或者宕机则会造成缓存数据的丢失,所以在实际项目开发中,开发者期望使用 Redis 或者 Memcached 等第三方框架对缓存数据进行统一管理与维护,附带的好处就是可以将缓存系统与项目解耦,并且可以通过以上这些框架提供的图形化管理工具更加方便的操作缓存数据,当然也可以使用比较“原始”的 telnet 命令操作缓存数据。由于 Redis 框架本身就支持数据持久化功能,因此本书选用 Redis 框架对流程定义实体对象进行缓存,Java 通常使用 Jedis 类库操作 Redis 实例,所以首先需要获取到 Jedis 程序包,本书中使用的 Jedis 程序包为 jedis.2.8.1。

8.6.1 自定义缓存处理类

接下来自定义一个缓存处理类并实现 DeploymentCache 接口,该类相关定义如代码清单 8-4 所示。

代码清单 8-4 ShareniuProcessDefinitionCache.java

```

1 public class ShareniuProcessDefinitionCache implements
2   DeploymentCache<ProcessDefinitionEntity> {
3   // 实例化 Jedis 类
4   Jedis jedis = new Jedis("localhost", 6379);
5   public ProcessDefinitionEntity get(String id) {
6       // 获取数据
7       byte[] bs = jedis.get(id.getBytes());
8       if (bs == null)
9           return null;
10      // 将二进制数据转换为 ProcessDefinitionEntity 对象
11      Object object = toObject(bs);
12      if (object == null)
13          return null;
14      ProcessDefinitionEntity pdf = (ProcessDefinitionEntity) object;
15      return pdf;
16  }
17  public void add(String id, ProcessDefinitionEntity object) {
18      // 添加到缓存, 因为 value 为 object 对象, 所以需要将该对象转化为二进制进行存储
19      jedis.set(id.getBytes(), toByteArray(object));
20  }
21  public void remove(String id) {
22      // 删除指定的 key
23      jedis.del(id.getBytes());
24  }
25  public void clear() {
26      // 清除所有数据, 这里不提供默认实现, 避免误操作
27  }
28  public static byte[] toByteArray(Object obj) { // 对象转化为 byte[]
29      byte[] bytes = null;
30      ByteArrayOutputStream bos = new ByteArrayOutputStream();
31      try {
32          ObjectOutputStream oos = new ObjectOutputStream(bos);
33          oos.writeObject(obj);
34          oos.flush();
35          bytes = bos.toByteArray();
36          oos.close();
37          bos.close();
38      } catch (IOException ex) {
39      }
40      return bytes;
41  }
42  public static Object toObject(byte[] bytes) { // 数组转对象
43      Object obj = null;
44      try {
45          ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
46          ObjectInputStream ois = new ObjectInputStream(bis);
47          obj = ois.readObject();
48          ois.close();
49          bis.close();

```

```
50     } catch (Exception ex) {  
51     }  
52     return obj; "usertask" name = "usertask"  
53     }  
54 }
```

在上述代码中,第1~2行定义了 `ShareniuProcessDefinitionCache` 类并实现 `DeploymentCache` 接口,本案例使用 Redis 键值对的方式操作缓存数据,存储的 key 为流程定义 id, value 为 `ProcessDefinitionEntity` 实例对象,由于 Redis 的键和值都支持使用二进制安全字符串,因此存储 Java 对象不是问题,前提是必须对 Java 对象(需要实现 `Serializable` 接口)进行序列化和反序列化。

第4行实例化 `Jedis` 类,程序可以通过该实例对象操作 Redis。第5~16行实现 `get` 方法的处理逻辑,其中第7行根据 id 参数从 Redis 中获取值,然后第11~15行调用 `toObject` 方法将获取到的值转化为 `ProcessDefinitionEntity` 实例对象并返回。

第17~20行实现了缓存的添加逻辑,因为 `DeploymentCache` 接口中没有定义数据的更新方法,因此需要在这里实现数据的更新操作。

第25行定义的 `clear` 方法该案例不提供实现,以防止客户端误操作造成缓存崩溃。

第28~41行定义的 `toByteArray` 方法用于将 `Object` 对象转化为 `byte[]`。第42~53行定义的 `toObject` 方法用于将 `byte[]` 转化为 `Object` 对象。

扩展

本案例中使用键值对的方式操作缓存数据,在实际项目开发中,也可以使用 Redis 框架中提供的其他数据结构进行操作。

8.6.2 验证自定义缓存处理类

接下来需要做的工作就是将 `ShareniuProcessDefinitionCache` 类注入流程引擎配置类,如代码清单 8-5 所示。

代码清单 8-5 activiti.cfg.xml

```
1 <bean id = "processEngineConfiguration"  
2     class = "org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">  
3     <!-- 通过 processDefinitionCache 属性注入自定义的缓存处理类 -->  
4     <property name = "processDefinitionCache" ref = "a"></property>  
5 </bean>  
6 <bean id = "a" class = "com.shareniu.chapter8.ShareniuProcessDefinitionCache"/>
```

在上述代码中,第4行通过设置 `processDefinitionCache` 开关属性进行自定义缓存处理类的注入。上述操作执行完毕可以部署任意一个流程文档,然后打开 Redis 操作界面,如果成功则如图 8-3 所示。

关于 `bpmnModelCache` 以及 `knowledgeBaseCache` 的缓存的相关知识,可结合该案例自行学习。

```
127.0.0.1:6379> KEYS *  
1) "shareniu:4:192504"  
127.0.0.1:6379>
```

图 8-3 Redis 中的缓存数据

8.7 Activiti 新特性之节点缓存

谈起节点缓存,可能会有这样的疑问:节点缓存的意义何在,什么是节点缓存,哪些节点支持缓存,节点缓存的值在哪里配置,节点缓存的数据格式又是什么。举一个常见的应用场景,例如在流程文档中,通常任务节点的使用会非常频繁,如果流程文档部署之后,流程实例运行了一段时间突然发现任务节点的名称、分类或者处理人需要修改,这时该怎么办呢?最简单的方法就是直接修改流程文档中不合理的地方然后再次部署流程文档并启动新的流程实例。但是流程文档的部署涉及流程文档的解析,而解析流程文档是非常昂贵的操作,仅仅是为了修改任务节点的部分信息就重新部署流程文档,难免有点得不偿失,如果需要频繁修改任务节点的信息,难道每次都要重新部署流程文档吗?显然该方案很不合理。

重新部署流程文档的方法只能解燃眉之急,并不能从根本上解决问题,该方式终究不是一个最优的解决方案。基于上述问题,节点缓存类 `ProcessDefinitionInfoCache` 应运而生,该类负责管理流程文档中节点(任务节点、服务任务节点等)的定义信息,该缓存类同其他缓存类一样需要交给 `DeploymentManager` 类进行管理,其内部使用 `Map` 集合管理缓存数据,该类虽然解决了缓存节点定义信息的问题,但是对于开发人员来说还是存在如下两个弊端。

(1) 缓存的插入以及更新问题。

节点缓存类中并没有提供更新节点缓存的方法,仅仅是在流程文档部署的时候才会对节点缓存数据进行初始化,如果启动流程实例的同时,缓存的流程定义实体对象的数据丢失,那么引擎会再次执行流程文档的部署操作,从而引起节点缓存重新添加,该过程可以参考第 13.2 节。

(2) 缓存数据持久化。

`Map` 集合中的数据存储在内存中,系统重启或者宕机,则缓存丢失。

注意

Activiti 在 5.19.0 版本引入节点缓存功能。

8.8 节点缓存实战

在设计流程文档的时候,可以为任务节点或者 `process` 元素添加节点缓存值,这样流程文档部署时引擎会将这些扩展的属性值添加到 `ACT_PROCDEF_INFO` 表中,下面定义一个流程文档并为其配置节点缓存值,然后进行部署,以观察 `ACT_PROCDEF_INFO` 表中的数据变化,相关实现如代码清单 8-6 所示。

代码清单 8-6 localization.bpmn20.xml

```

1 <process id="operationProcess" name="process" isExecutable="true">
2   <extensionElements>
3     <activiti:localization key="shareniu" id="shareniuProcess"
4       name="shareniuProcess" locale="shareniuProcess">
5       <documentation>us description</documentation>

```

```

6      </activiti:localization>
7  </extensionElements>
8  <userTask id = "usertask1" name = "usertask1">
9      <extensionElements>
10         <activiti:localization key = "shareniu" id = "US" name = "shareniuUsertask1"
11             locale = "shareniuUsertask1">
12             <documentation> usertask1 </documentation>
13         </activiti:localization>
14     </extensionElements>
15 </userTask>
16 <startEvent id = "timerstartevent1" name = "start"></startEvent>
17 <sequenceFlow id = "flow5" sourceRef = "timerstartevent1" targetRef = "usertask1">
</sequenceFlow>
18 <userTask id = "usertask2" name = "usertask2">
19     <extensionElements>
20         <activiti:bpmnn id = "US" name = "shareniuUsertask2" locale = "shareniuUsertask2">
21             <documentation> usertask2 </documentation>
22         </activiti:bpmnn>
23     </extensionElements>
24 </userTask>
25 <sequenceFlow id = "flow6" sourceRef = "usertask1" targetRef = "usertask2"></sequenceFlow>
26 <endEvent id = "endevent1" name = "End"></endEvent>
27 <sequenceFlow id = "flow7" sourceRef = "usertask2" targetRef = "endevent1"></sequenceFlow>
28 </process>

```

在上述代码中,第2~7行使用 `activiti:localization` 元素为 `process` 元素添加了节点缓存,其中第3~4行定义了 `key`、`id`、`name`、`locale` 四个属性,第9~14行为 `usertask1` 元素添加了节点缓存,第19~23行为 `usertask2` 元素添加了节点缓存。

注意

第3行和第10行使用的元素为 `activiti:localization`,第20行使用的元素为 `activiti:bpmnn`。

由于节点缓存功能引擎默认是关闭的,因此要想使用该功能,则必须开启,如代码清单8-7所示。

代码清单 8-7 activiti2.cfg.xml

```

1 <bean id = "processEngineConfiguration"
2     class = "org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <property name = "enableProcessDefinitionInfoCache" value = "true" />
4 </bean>

```

在上述代码中,第3行设置 `enableProcessDefinitionInfoCache` 开关属性值为 `true`。接下来要做的工作就是部署上述流程文档进而观察数据库 `ACT_PROCDEF_INFO` 表相应的数据变化如图8-4所示。

ID	PROC_DEF_ID	REV	INFO_JSON_ID
215005	operationProcess:6:215004		1 215006

图 8-4 ACT_PROCDEF_INFO 表中的数据

扩展

ACT_PROCDEF_INFO 表中的 INFO_JSON_ID_列关联 ACT_GE_BYTEARRAY 表中的主键 ID_列。

紧接着查看 ACT_GE_BYTEARRAY 表中 ID_为 215006 的数据,如代码清单 8-8 所示。

代码清单 8-8 ACT_GE_BYTEARRAY 表中 ID_为 215006 的数据

```

1 {
2   "localization": {
3     "shareniuProcess": {
4       "operationProcess": {
5         "name": "shareniuProcess",
6         "description": "us description"
7       }
8     },
9     "shareniuUsertask1": {
10      "usertask1": {
11        "name": "shareniuUsertask1",
12        "description": "usertask1"
13      }
14    }
15  }
16 }

```

观察上述代码并结合上文定义的流程文档,可以得出如下结论。

(1) 在上述代码中,第 3 行 shareniuProcess 值为代码清单 8-6 第 4 行定义的 locale 属性值。

(2) 在上述代码中,第 4 行 operationProcess 值为代码清单 8-6 第 1 行定义的 id 属性值。

(3) 在上述代码中,第 9 行 shareniuUsertask1 值为代码清单 8-6 第 11 行定义的 locale 属性值。

(4) 在上述代码中,第 10 行 usertask1 值为代码清单 8-6 第 8 行定义的 id 属性值。

(5) 在代码清单 8-6 中,第 19~23 行为 usertask2 元素添加的节点缓存并没有存储在 ACT_GE_BYTEARRAY 表中。换言之,只有在 activiti:localization 元素中定义的节点缓存数据才会添加到数据库。

8.9 节点缓存原理

Activiti 是如何处理节点缓存数据的呢? 又是如何将流程文档中定义的节点缓存数据解析出来并存储到数据库的呢? 本节深入讲解。

8.9.1 初始化节点缓存数据

7.3 节中 addDefinitionInfoToCache 方法用于处理流程文档中定义的节点的缓存数据, 该方法的定义如代码清单 8-9 所示。

代码清单 8-9 BpmnDeployer.java

```

1  protected void addDefinitionInfoToCache(ProcessDefinitionEntity processDefinition,
2  ProcessEngineConfigurationImpl processEngineConfiguration, CommandContext commandContext) {
3      if (processEngineConfiguration.isEnableProcessDefinitionInfoCache() == false) {
4          return; //如果没有开启流程定义节点缓存,则不会进行如下的处理,直接返回
5      }
6      DeploymentManager deploymentManager = processEngineConfiguration.getDeploymentManager();
7      ProcessDefinitionInfoEntityManager definitionInfoEntityManager =
8      commandContext.getProcessDefinitionInfoEntityManager();
9      ObjectMapper objectMapper = commandContext.getProcessEngineConfiguration().
10      getObjectMapper();
11      //根据 processDefinition 对象的 id 值,从 ACT_PROCDEF_INFO 表中查询数据
12      ProcessDefinitionInfoEntity definitionInfoEntity = definitionInfoEntityManager.
13      findProcessDefinitionInfoByProcessDefinitionId(processDefinition.getId());
14      ObjectNode infoNode = null;
15      if (definitionInfoEntity != null && definitionInfoEntity.getInfoJsonId() != null) {
16          //从 ACT_GE_BYTEARRAY 表中获取数据
17          byte[] infoBytes =
18          definitionInfoEntityManager.findInfoJsonById(definitionInfoEntity.getInfoJsonId());
19          if (infoBytes != null) { //如果获取到数据
20              try {
21                  infoNode = (ObjectNode) objectMapper.readTree(infoBytes);
22                  //将 infoBytes 转换为 infoNode
23              } catch (Exception e) {
24                  throw new ActivitiException("Error" );
25              }
26          }
27          //实例化 ProcessDefinitionInfoCacheObject 类
28          ProcessDefinitionInfoCacheObject definitionCacheObject =
29          new ProcessDefinitionInfoCacheObject();
30          if (definitionInfoEntity == null) {
31              definitionCacheObject.setRevision(0);
32          } else {
33              definitionCacheObject.setId(definitionInfoEntity.getId());
34              definitionCacheObject.setRevision(definitionInfoEntity.getRevision());
35          }
36          if (infoNode == null) {
37              infoNode = objectMapper.createObjectNode(); //创建根节点,保证 infoNode 对象不为空
38          }
39          definitionCacheObject.setInfoNode(infoNode);
40          deploymentManager.getProcessDefinitionInfoCache().add(processDefinition.getId(),
41          definitionCacheObject); //将数据添加到缓存中
42      }

```

(1) 在上述代码中,第 3 行判断 enableProcessDefinitionInfoCache 开关属性值,如果该值为 false,则该方法直接返回,通过该操作可以看出,要想使用节点缓存功能,必须设置该值为 true。

(2) 第 6~10 行获取 deploymentManager 对象、definitionInfoEntityManager 对象、objectMapper 对象。

(3) 第 12~13 行根据 processDefinition 对象的 id 值,从 ACT_PROCDEF_INFO 表中查询数据,如果查询到数据,则执行第 17~18 行代码从 ACT_GE_BYTEARRAY 表中获取数据。

(4) 第 21 行将查询到的 infoBytes 数据转化为 infoNode 对象。

(5) 第 28~39 行实例化 ProcessDefinitionInfoCacheObject 类,并为其填充属性值。

(6) 第 40~41 行将 definitionCacheObject 对象添加到节点缓存中。

(7) 上述的一系列操作仅仅是根据 processDefinition 对象的 id 值从数据库查询数据并将其设置到缓存中,并没有对流程文档中定义的节点缓存进行解析和入库,引擎为何这样设计,出于何种考虑? 下面分析第 7.3 节中 createLocalizationValues 方法实现的功能。

8.9.2 更新节点缓存

createLocalizationValues() 方法的具体实现如代码清单 8-10 所示。

代码清单 8-10 BpmnDeployer.java

```

1  protected void createLocalizationValues(String processDefinitionId, Process process) {
2      if (process == null) return;
3      CommandContext commandContext = Context.getCommandContext(); //获取命令上下文对象
4      DynamicBpmnService dynamicBpmnService =
5      commandContext.getProcessEngineConfiguration().getDynamicBpmnService();
6      //根据流程定义 Id 从 ACT_PROCDEF_INFO 表中查询数据
7      ObjectNode infoNode = dynamicBpmnService.getProcessDefinitionInfo(processDefinitionId);
8      boolean localizationValuesChanged = false; //记录节点信息是否发生变化
9      List<ExtensionElement> localizationElements =
10     process.getExtensionElements().get("localization");
11     If (localizationElements != null){
12         //如果 extensionElements 元素中有 localization 子元素则开始获取
13         for (ExtensionElement localizationElement : localizationElements) { //循环遍历元素
14             //通过下面的操作可以看出命名空间必须是 activiti 才可以
15             if ("activiti".equals(localizationElement.getNamespacePrefix())) {
16                 String locale = localizationElement.getAttributeValue(null, "locale");
17                 //获取 locale 性
18                 String name = localizationElement.getAttributeValue(null, "name");
19                 //获取 name 值
20                 String documentation = null; //获取 documentation 文本值
21                 List<ExtensionElement> documentationElements =
22                 localizationElement.getChildElements().get("documentation");
23                 if (documentationElements != null) { //如果查找到就 break
24                     for (ExtensionElement documentationElement : documentationElements) {
25                         documentation = StringUtils.trimToNull(documentationElement.getElementText());
26                     }
27                     break; //如果配置有多个 documentation 元素只会执行一次
28                 }
29             }
30         }
31     }
32 }

```

55 }

接返回。

(2) 第 3~5 行获取 `commandContext`、`dynamicBpmnService`。

终会调用 `GetProcessDefinitionInfoCmd` 命令类从缓存中获取数据。

以看出只有 localization 元素才可以被引擎解析,其他的元素引擎不会处理。

(5) 如果 localizationElements 集合不为空,则开始执行如下处理逻辑。

如果是则继续处理,否则不会进行如下操作。

- 第 15~16 行分别获取 locale、name 属性值。

- 第 18~25 行获取子元素 documentation 值。

- 第 27 行调用 `isEqualToCurrentLocalizationValue` 方法进行 `name` 数据(缓存中的数

据与流程文档中定义的数据)的比对工作,如果两者数据不一致则执行第 30 行更新 infoNode 对象(将两者数据合并)。

- 第 33 行调用 `isEqualToCurrentLocalizationValue` 方法进行 `description(documentation)` 数据的比对工作,如果两者数据不一致则执行第 35 行更新 infoNode 对象(将两者数据合并)。
- 第 39 行 `break` 终止循环,该操作隐含透漏一个信息,流程文档中同一个元素可以定义多个并行的 `localization` 元素,但是该方法处理的时候只会解析第一个 `localization` 元素。

(6) 第 44 行调用 `localizeFlowElements` 方法,查找并比对 `process` 对象中的子元素对象与相应的节点缓存数据是否一致,如果不一致,则执行更新操作,`localizeFlowElements` 方法的处理过程可以参考该案例的讲解。

(7) 第 48 行调用 `localizeDataObjectElements` 方法对比 `dataObject` 元素与相应的缓存数据是否一致,只要两者数据不一致,则执行更新操作。

(8) 如果 `localizationValuesChanged` 变量为 `true`,则执行第 53 行代码,将 `infoNode` 数据更新到数据库。

通过 `createLocalizationValues` 方法的处理逻辑可以看出,activiti:localization 元素暂时支持的属性有 `id`、`name` 以及 `locale`,支持的子元素只有 `documentation`,只有 `process`、`userTask`、`subProcess`、`dataObject` 等元素支持节点缓存功能。

注意

第 53 行调用的 `saveProcessDefinitionInfo` 方法只会将数据更新到数据库,并不会更新缓存,很显然这是 Activiti 的一个 Bug。

8.9.3 节点缓存架构

接下来,对上文讲解的 `addDefinitionInfoToCache` 方法和 `createLocalizationValues` 方法所实现的功能进行总结,首先使用图 8-5 对上述两个方法的处理流程进行通俗易懂的描述。

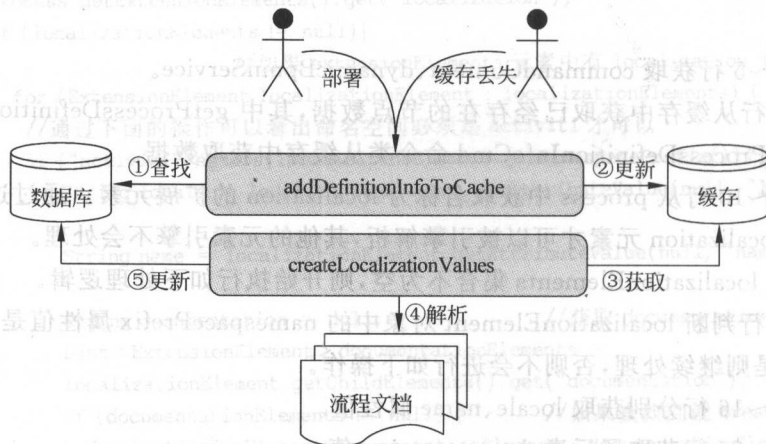


图 8-5 节点缓存处理

如果部署流程文档或者运转流程实例时流程定义缓存丢失,则需要执行上图中的两个方法,首先 `addDefinitionInfoToCache` 方法会从数据库中查找节点缓存数据,不管是否查询到都会将其更新到缓存中,看到这里的处理逻辑,可能会有疑问:如果从数据库中并没有查找到数据,怎么还需要更新缓存呢?首先要明白一个问题,缓存中的数据是 `ProcessDefinitionInfoCacheObject` 实例对象,因为 `createLocalizationValues` 方法在处理过程中并没有对该对象进行非空校验,因此即使缓存中没有数据,也需要实例化一个 `ProcessDefinitionInfoCacheObject` 对象进行存储,这样才可以保证 `createLocalizationValues` 方法执行时不会报错,这样设计归根结底是 Activiti 中的一个漏洞导致的,因此使用的时候需要多加小心。`createLocalizationValues` 方法的处理逻辑如下。

- (1) 从缓存中获取数据。
- (2) 解析流程文档中定义的缓存数据。
- (3) 对比以上两者数据,如果两者数据内容不一致,则首先将两者数据进行合并,然后将合并之后的数据更新到数据库。

8.9.4 节点缓存使用误区

在实现自定义节点缓存类之前,首先思考如下几个问题。

- (1) 自定义节点缓存类如何替换引擎默认的节点缓存类。
能否直接设置 `ProcessEngineConfigurationImpl` 实例对象中的 `processDefinitionInfoCache` 开关属性值呢?但是很遗憾,`ProcessEngineConfigurationImpl` 类中并没有发现任何设置 `processDefinitionInfoCache` 属性值的方法,这个时候应该怎么做呢?既然流程引擎配置类没有提供该属性的设置方法,开发人员可以自定义一个引擎配置类并继承 `StandaloneProcessEngineConfiguration` 类从而完成对 `processDefinitionInfoCache` 属性赋值操作。

- (2) 命令执行器如何注入自定义节点缓存类。

在 `ProcessDefinitionInfoCache` 类的构造方法中需要一个输入参数,该参数的类型为 `CommandExecutor`,因此自定义节点缓存类继承 `ProcessDefinitionInfoCache` 类的同时必须显式调用其父类的构造方法,为了确保父类不被破坏,能继续独立运行,所以需想尽一切办法获取 `CommandExecutor` 实例对象并将其传递给父类,形如 `public ShareniuProcessDefinitionInfoCache(CommandExecutor ce) {super(ce);}`,那么问题来了,如果客户端使用 Spring 进行 bean 的实例化工作,`CommandExecutor` 实例对象作为流程引擎配置类中的属性存在,该如何获取?因为 `CommandExecutor` 类的实例化工作完全由引擎内部实现,很显然,通过配置文件方式将该自定义节点缓存类注入到 `ProcessEngineConfigurationImpl` 类中有点不大可能实现,既然该方式不太容易实现,不妨换一个思路,第 2.5 节中详细讲解了 Activiti 配置器,在此不妨一试,因为 `CommandExecutor` 实例可以通过流程引擎配置类实例对象直接获取。

- (3) `ProcessDefinitionInfoCacheObject` 类。

因为 `ProcessDefinitionInfoCache` 类默认将节点缓存数据封装到 `ProcessDefinitionInfoCacheObject` 类中,该类封装了流程定义 id、节点版本信息以及节点缓存数据对应的 `ObjectNode` 对象,形如 `{"bpmn":{"usertask1":{"userTaskName":"shareniu"}}`。由于本书使用 Redis 缓存节点定义信息,因此需要缓存的对象必须实现 Java 中的 `Serializable` 接口,Activiti 在定义 `ProcessDefinitionInfoCacheObject` 类时并没有实现 `Serializable` 接口,所

以使用 Redis 存储 ProcessDefinitionInfoCacheObject 实例对象肯定会报错,因此需要自定义一个缓存数据存储类,并实现该类实例对象与 ProcessDefinitionInfoCacheObject 实例对象之间可以相互转换。

(4) 上述讲解的 createLocalizationValues 方法只会更新数据库中的节点缓存数据而不会更新缓存中的数据,这也是 Activiti 的一个 Bug。

节点缓存类的设计完全不符合 Activiti 的一贯风格,Activiti 的一贯做法就是将功能抽象为接口并提供默认实现类,但是 Activiti 并没有将节点缓存处理类 ProcessDefinitionInfoCache 设计为接口,反倒只是一个普通的类,而且需要缓存的对象并没有实现序列化接口,如果客户端需要自定义节点缓存类则必须继承 ProcessDefinitionInfoCache 类,但是该类中并没有定义无参构造方法,因此子类必须在自身构造方法中获取命令执行器 commandExecutor,这样的设计大大增加了客户端操作的复杂度,而且自定义节点缓存类需要通过 processDefinitionInfoCache 属性注入 ProcessEngineConfigurationImpl 类,但是引擎并没有提供任何设置 processDefinitionInfoCache 属性的方法,综上所述实现自定义节点缓存类,对于开发人员来说根本就是一次代码重写,上面所陈述的一系列问题都需要解决方案。

既然上面提出的一系列问题 Activiti 并没有提供解决方案,所以本节只能通过扩展源码的方式实现相应的功能。能不能使用修改源码的方式解决上述问题?当然可以,但是扩展源码的方式比直接修改源码的方式要优雅一点,因为如果直接修改源码并对 Activiti 中不合理的地方进行改造,则 Activiti 框架需要升级时,修改的代码就需要迁移到新版本,这个工作量也不小,而扩展源码则不会出现类似这样的问题,也可以说是一种更优雅、更灵活的方案。

8.10 自定义节点缓存实战

8.10.1 自定义节点缓存类

首先自定义一个节点缓存类,相关实现如代码清单 8-11 所示。

代码清单 8-11 ShareniuProcessDefinitionInfoCache.java

```
1 public class ShareniuProcessDefinitionInfoCache extends ProcessDefinitionInfoCache {
2     public static class ProcessDefinitionInfoCacheObjectVo implements Serializable {
3         protected String id; //id
4         protected int revision; //版本
5         protected String infoNode;
6     }
7     protected CommandExecutor commandExecutor;
8     public void setCommandExecutor(CommandExecutor commandExecutor) {
9         super.commandExecutor = commandExecutor;
10        this.commandExecutor = commandExecutor;
11    }
12    Jedis jedis = new Jedis("127.0.0.1");
13    String prex = "shareniuNodeCache:";
14    public void add(String id, ProcessDefinitionInfoCacheObject obj) {
15        // redis 获取
```

```

16 byte[] bs = jedis.get((prex + id).getBytes()); // 从缓存中获取数据
17 Object object = null;
18 // 实例化自定义的类并完成与 ProcessDefinitionInfoCacheObject 类进行相互转化
19 ProcessDefinitionInfoCacheObjectVo v = new ProcessDefinitionInfoCacheObjectVo();
20 ObjectMapper objectMapper = new ObjectMapper();
21 ObjectNode dataBaseInfoNode = obj.getInfoNode(); // 获取节点的缓存数据
22 try {
23     ObjectNode redisObjectNode = objectMapper.createObjectNode(); // 如果不存在则实例化
24     String redisStr = null;
25     if (bs != null) {
26         // 将二进制流转化为对象
27         object = ShareniuProcessDefinitionCache.toObject(bs);
28         // 对象转化, 因为添加时类型为 ProcessDefinitionInfoCacheObjectVo, 所以存在值转化不
29         // 会报错
30         ProcessDefinitionInfoCacheObjectVo pdf = (ProcessDefinitionInfoCacheObjectVo) object;
31         redisStr = pdf.getInfoNode(); // 获取缓存中的数据并转化为 ObjectNode 类型的对象
32         redisObjectNode = (ObjectNode) objectMapper.readTree(redisStr);
33     }
34     v.setId(obj.getId()); // 设置 id
35     // 如果添加数据时发现 redis 有值, 则将新的值与已经存在的值合并, 并填充对象的属性值
36     dataBaseInfoNode.putAll(redisObjectNode);
37     String finall = objectMapper.writeValueAsString(dataBaseInfoNode);
38     v.setInfoNode(finall);
39     v.setRevision(obj.getRevision());
40     // 将对象转化为 byte[] 存储到 redis
41     jedis.set((prex + id).getBytes(), ShareniuProcessDefinitionCache.toByteArray(v));
42 } catch (Exception e) {}
43 }
44 public ProcessDefinitionInfoCacheObject get(final String id) {
45     // 从缓存中获取数据
46     byte[] bs = jedis.get((prex + id).getBytes());
47     if (bs == null) return null;
48     // 将二进制数据转换为 object 对象防止为空
49     Object object = ShareniuProcessDefinitionCache.toObject(bs);
50     if (object == null) return null;
51     ProcessDefinitionInfoCacheObjectVo pdf = (ProcessDefinitionInfoCacheObjectVo) object;
52     // 将从缓存中获取到的对象进行转换并返回
53     ProcessDefinitionInfoCacheObject v = new ProcessDefinitionInfoCacheObject();
54     v.setId(pdf.getId());
55     ObjectMapper objectMapper = new ObjectMapper();
56     ObjectNode infoNode = null;
57     try {
58         if (objectMapper.readTree(pdf.getInfoNode()) != null) {
59             infoNode = (ObjectNode) objectMapper.readTree(pdf.getInfoNode());
60         }
61         v.setInfoNode(infoNode);
62         v.setRevision(pdf.getRevision());
63         return v;
64     } catch (IOException e) {}
65     return null;

```



```

66 }
67 public void clear() { //该方法不提供实现 }
68 public ShareniuProcessDefinitionInfoCache(CommandExecutor commandExecutor) {
69     super(commandExecutor);
70 }

```

仅从代码量上就能看出自定义节点缓存类的处理逻辑非常复杂,将其处理逻辑梳理如下。

(1) 缓存数据存储类。

首先第 2~6 行定义了一个静态内部类 ProcessDefinitionInfoCacheObjectVo,该类主要是为了解决 ProcessDefinitionInfoCacheObjectVo 实例对象与 ProcessDefinitionInfoCacheObject 实例对象的相互转换,由于 ProcessDefinitionInfoCacheObject 类中的 infoNode 属性为 ObjectNode 类型,而 ObjectNode 实例对象不能进行序列化操作,因此需要将自定义类中的 infoNode 属性定义为 String 类型,两者的数据是可以相互转换的。

(2) 添加缓存数据。

第 14~43 行定义 add 方法主要用于添加缓存数据,处理逻辑很简单,首先第 16 行获取 Redis 中已经存在的缓存数据,第 21 行通过 obj 获取节点的缓存数据,如果缓存中存在数据,则将两者数据进行合并、添加到缓存。具体的实现细节涉及了对象与 byte[] 数组之间的相互转换,可以参考上文的讲解。这里需要注意,只要确保 add 方法中操作的数据不为空即可,因为引擎第一次获取数据时,如果没有获取到值,则程序就会报错(Activiti 并没有对其进行非空校验,也算是 Activiti 少有的缺陷之一),当然在实际项目开发中,可以直接使用 JSON 生成工具生成整个节点缓存数据,然后通过 Redis 客户端进行添加,既方便又省事。

(3) 获取缓存数据。

第 44~66 行,根据流程定义 id 从缓存中获取数据,如果没有获取到数据则第 50 行直接返回空,否则进行数据的转化以及组装操作,最后返回结果。

8.10.2 修复 Activiti 节点缓存不更新 Bug

createLocalizationValues 方法仅仅是更新了数据库中的节点缓存数据,并没有更新缓存中的数据,因此需要修复该 Bug。既然 createLocalizationValues 方法位于 BpmnDeployer 类中,那就可以自定义一个部署器并继承 BpmnDeployer 类,然后重新实现该方法的处理逻辑,只需要在代码清单 8-10 第 53 行之后添加更新缓存的逻辑即可,由于本书篇幅有限,下面仅罗列出更新缓存的实现代码,如代码清单 8-12 所示。

代码清单 8-12 ShareniuDeployer.java

```

1 public class ShareniuDeployer extends BpmnDeployer {
2     protected void createLocalizationValues(String processDefinitionId, Process process) {
3         ProcessEngineConfigurationImpl processEngineConfiguration =
4             Context.getProcessEngineConfiguration();
5         DeploymentManager deploymentManager = processEngineConfiguration.getDeploymentManager();
6         ProcessDefinitionInfoCacheObject definitionCacheObject = new

```

```

7     ProcessDefinitionInfoCacheObject();
8     definitionCacheObject.setInfoNode(infoNode);
9     if (localizationValuesChanged) {
10        deploymentManager.getProcessDefinitionInfoCache().add(processDefinitionId,
11        definitionCacheObject);
12    }
13 }
14 }

```

在上述代码中,第3~4行获取 processEngineConfiguration 对象,第5行获取 deploymentManager 对象,第6~7行实例化 ProcessDefinitionInfoCacheObject 类,第10~11行将 definitionCacheObject 对象添加到缓存中。

8.10.3 扩展引擎配置类功能

接下来,自定义引擎配置类并继承 StandaloneProcessEngineConfiguration 类,然后通过动态属性注入方式为其注入 processDefinitionInfoCache 属性值,相关实现如代码清单 8-13 所示。

代码清单 8-13 ShareniumStandaloneProcessEngineConfiguration.java

```

1 public class ShareniumStandaloneProcessEngineConfiguration extends
2     StandaloneProcessEngineConfiguration {
3     // 设置 processDefinitionInfoCache 对象的值
4     public void setProcessDefinitionInfoCache(ProcessDefinitionInfoCache cache) {
5         super.processDefinitionInfoCache = cache;
6     }
7 }

```

在上述代码中,第4行定义了 setProcessDefinitionInfoCache 方法,该方法用于设置 processDefinitionInfoCache 值。

8.10.4 配置器高级用法

下面需要自定义一个配置器,然后将 ShareniumProcessDefinitionInfoCache 类注入自定义引擎配置类,相关实现如代码清单 8-14 所示。

代码清单 8-14 ShareniumProcessEngineConfigurator.java

```

1 public class ShareniumProcessEngineConfigurator implements ProcessEngineConfigurator{
2     ShareniumStandaloneProcessEngineConfiguration a;
3     ShareniumProcessDefinitionInfoCache cache;
4     public void beforeInit(ProcessEngineConfigurationImpl pc) {
5         a = (ShareniumStandaloneProcessEngineConfiguration)pc;
6         cache = new ShareniumProcessDefinitionInfoCache(null);
7         a.setProcessDefinitionInfoCache(cache);
8     }
9     public void configure(ProcessEngineConfigurationImpl pc) {

```

```

10     a = (ShareniuStandaloneProcessEngineConfiguration)pc;
11     if (cache == null) cache = new ShareniuProcessDefinitionInfoCache(a.getCommandExecutor());
12     cache.setCommandExecutor(a.getCommandExecutor());
13     a.setProcessDefinitionInfoCache(cache);
14 }
15 }

```

通过上面的代码可知, beforeInit 方法以及 configure 方法都需要设置 ShareniuProcessDefinitionInfoCache 值, 是不是多此一举呢? 再次回顾一下 ProcessEngineConfigurationImpl 类中 init 方法的初始化先后顺序, 如下所示。

(1) configuratorsBeforeInit(), 该方法负责遍历所有的配置器实例对象并依次调用该实例对象中的 beforeInit 方法。

(2) initCommandExecutors(), 初始化命令调度者、命令执行器等。该过程可以参考第 12.2 节。

(3) initDeployers(), 该方法内部实现会初始化各种缓存处理类, 如果开发人员配置有自定义缓存处理类, 则优先使用。

(4) configuratorsAfterInit(), 该方法负责遍历所有的配置器实例对象并依次调用该实例对象中的 configure 方法。

如果第 4 行定义的 beforeInit 方法中没有注入自定义节点缓存类, 则 initDeployers 方法就直接使用系统内置的节点缓存类, 所以该步骤非常有必要且不可缺少, 该步骤明确告诉流程引擎客户端使用自定义节点缓存类。

由于 initCommandExecutors 方法在 beforeInit 方法执行之后才开始调用, 所以命令执行器实例对象在 beforeInit 方法中是无法获取的, 因此需要灵活变通, 首先将自定义节点缓存类注入引擎配置类, 然后将命令执行器实例对象设置为空, 最终在 configure 方法中获取命令执行器实例对象并重新将其设置到自定义节点缓存类实例对象中。

强调

beforeInit 和 configure 方法操作的 cache 必须为同一个, 否则后续使用时可能出问题。

8.10.5 使用自定义节点缓存类

接下来需要做的工作就是将上述自定义的配置器、部署器注入流程引擎配置类, 如代码清单 8-15 所示。

代码清单 8-15 activiti3.cfg.xml

```

1 <bean id="processEngineConfiguration"
2 class="com.shareniu.chapter8.ShareniuStandaloneProcessEngineConfiguration">
3   <property name="enableProcessDefinitionInfoCache" value="true" />
4   <property name="bpmnDeployer" ref="shareniuBpmnDeployer"></property>
5   <property name="configurators">
6     <list>

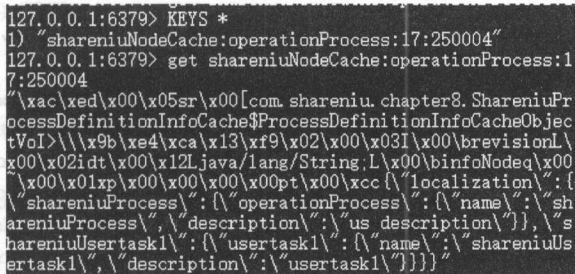
```

```

7  <!-- 自定义配置器 -->
8  <bean class = "com.shareniu.chapter8.ShareniuProcessEngineConfigurator"></bean>
9  </list>
10 </property>
11 </bean>
12 <bean id = "shareniuBpmnDeployer" class = "com.shareniu.chapter8.ShareniuDeployer"/>

```

在上述代码中,第3行开启节点缓存功能,第5~10行通过 configurators 开关属性设置配置器。接下来部署代码清单 8-6 中定义的流程文档进而观察 Redis 中相应的数据变化,如图 8-6 所示。



```

127.0.0.1:6379> KEYS *
1) "shareniuNodeCache:operationProcess:17:250004"
127.0.0.1:6379> get shareniuNodeCache:operationProcess:17:250004
{"name":"shareniuNodeCache:operationProcess:17:250004","description":"shareniuNodeCache:operationProcess:17:250004","userTaskId":"shareniuNodeCache:operationProcess:17:250004"}

```

图 8-6 Redis 中存储的节点缓存数据

8.11 任务节点缓存数据获取原理

上文浓墨重彩地讲解了自定义节点缓存类以及将其注入流程引擎配置类的过程,看到这里可能会有疑问,引擎在何地调用节点缓存数据呢? 流程引擎执行节点的时候,又是如何获取并解析节点缓存数据呢? 这也是接下来重点讲解的地方。任务节点的缓存信息是在任务需要执行的地方进行解析,例如任务的完成操作或者任务节点的入库操作,关于这一点可以参考 14.4 节,在此直接查找任务节点的行为类 UserTaskActivityBehavior 的执行方法 execute,如代码清单 8-16 所示。

代码清单 8-16 UserTaskActivityBehavior.java

```

1  public void execute(ActivityExecution execution) throws Exception {
2      ...//省略代码
3      if (Context.getProcessEngineConfiguration().isEnabledProcessDefinitionInfoCache()) {
4          ObjectNode taskElementProperties = Context.getBpmnOverrideElementProperties(userTaskId,
5              execution.getProcessDefinitionId());
6          activeNameExpression = getActiveValue(taskDefinition.getNameExpression(), "userTaskName",
7              taskElementProperties);
8          taskDefinition.setNameExpression(activeNameExpression);
9          ...//省略操作任务节点其他属性的代码
10 }

```

在上述代码中,第3行验证引擎是否开启了节点缓存功能,如果流程引擎开启了该功能,则第4~5行从缓存中查找任务节点的缓存数据,并对其进行解析和运用,第6~7行调

用 `getActiveValue` 方法从 `taskElementProperties` 对象中获取 `activeNameExpression` 值,第 8 行将其填充到 `taskDefinition` 对象中。接下来分析 `getBpmnOverrideElementProperties` 方法的处理逻辑。

8.11.1 获取任务节点缓存数据

`Context` 类中的 `getBpmnOverrideElementProperties` 方法首先根据任务节点的 `id` 值和 `processDefinitionId` 值从缓存中获取数据,相关实现如代码清单 8-17 所示。

代码清单 8-17 Context.java

```
1 public static ObjectNode getBpmnOverrideElementProperties(String id, String
2   processDefinitionId){
3   ObjectNode definitionInfoNode = getProcessDefinitionInfoNode(processDefinitionId);
4   ObjectNode elementProperties = null;
5   if (definitionInfoNode != null) {
6     elementProperties = getProcessEngineConfiguration().getDynamicBpmnService().
7       getBpmnElementProperties(id,definitionInfoNode);
8   }
9   return elementProperties;
10 }
```

`getBpmnOverrideElementProperties` 方法的处理逻辑分为如下两个重要部分。

(1) 上述代码中,第 3 行调用 `getProcessDefinitionInfoNode` 方法,并传入 `processDefinitionId` 参数值进行处理,该方法的具体实现如代码清单 8-18 所示。

代码清单 8-18 Context.java

```
1 protected static ObjectNode getProcessDefinitionInfoNode(String processDefinitionId) {
2   Map<String, ObjectNode> bpmnOverrideMap = getBpmnOverrideContext();
3   if (bpmnOverrideMap.containsKey(processDefinitionId) == false) {
4     ProcessDefinitionInfoCacheObject cacheObject =
5       getProcessEngineConfiguration().getDeploymentManager()
6       .getProcessDefinitionInfoCache().get(processDefinitionId);
7     addBpmnOverrideElement(processDefinitionId, cacheObject.getInfoNode());
8   }
9   return getBpmnOverrideContext().get(processDefinitionId);
10 }
```

在上述代码中,第 2 行调用 `getBpmnOverrideContext` 方法,该方法主要是为了确保 `Context` 类中的 `bpmnOverrideContextThreadLocal` 成员变量已经被初始化,第 3 行如果 `processDefinitionId` 不存在 `bpmnOverrideMap` 集合中,则第 4~7 行从缓存中查找该 `processDefinitionId` 对应的节点缓存数据。

(2) 第 5 行如果 `definitionInfoNode` 不为空,则第 6~7 行调用 `DynamicBpmnServiceImpl` 实例对象中的 `getBpmnElementProperties` 方法获取当前任务节点的缓存数据。

注意

bpmnOverrideContextThreadLocal 为 ThreadLocal 类型, ThreadLocal 是为了解决线程安全问题而设计的,该类内部维护一个 Map 集合,用于存储每一个线程变量的副本, Map 中的 key 为当前线程对象, value 为对应的线程变量副本,由于 key 是不能重复的,从而达到了线程安全的目的。

8.11.2 解析任务节点缓存数据

getBpmnElementProperties 方法获取当前任务节点对应的缓存数据,该方法的相关实现如代码清单 8-19 所示。

代码清单 8-19 DynamicBpmnServiceImpl.java

```
1 public ObjectNode getBpmnElementProperties(String id, ObjectNode infoNode) {  
2     ObjectNode propertiesNode = null;  
3     ObjectNode bpmnNode = getBpmnNode(infoNode);  
4     if (bpmnNode != null) {  
5         propertiesNode = (ObjectNode) bpmnNode.get(id);  
6     }  
7     return propertiesNode;  
8 }  
9 protected ObjectNode getBpmnNode(ObjectNode infoNode) {  
10     return (ObjectNode) infoNode.get("bpmn");  
11 }  
12 }
```

在上述代码中,第 3 行调用第 9 行定义的 getBpmnNode 方法进行处理,第 10 行从 infoNode 中获取 key 为“bpmn”的值,这个地方的处理有问题,因为引擎并不会解析 activiti:bpmnn 元素,所以获取该元素下配置的节点缓存数据显然不合理,不过没有关系,可以通过上文讲解的手动构造数据并更新缓存的方式使用该功能。第 4 行如果 bpmnNode 不为空,则将 id 值作为查询条件并从 bpmnNode 对象中获取该 id 值下面所有的数据集合并返回。

8.11.3 运用任务节点缓存数据

getActiveValue 方法主要用户获取当前任务节点的属性值并判断该属性值是否存在于节点缓存中,如果存在则开始使用,相关实现如代码清单 8-20 所示。

代码清单 8-20 UserTaskActivityBehavior.java

```
1 protected Expression getActiveValue(Expression originalValue, String propertyName, ObjectNode  
2 taskElementProperties) {  
3     Expression activeValue = originalValue;  
4     if (taskElementProperties != null) {  
5         JsonNode overrideValueNode = taskElementProperties.get(propertyName);  
6         if (overrideValueNode != null) {
```

```

7         if (overrideValueNode.isNull()) {
8             activeValue = null;
9         } else {
10            activeValue = Context.getProcessEngineConfiguration().getExpressionManager().
11                createExpression(overrideValueNode.asText());
12        }
13    }
14 }
15 return activeValue;
16 }

```

在上述代码中,第4行判断 `taskElementProperties` 参数值是否为空,如果该参数值为空,则说明当前任务节点不存在节点缓存数据,因此不会进行后续处理;如果该参数值不为空,则执行第5~13行代码,其中第5行根据 `propertyName` 从 `taskElementProperties` 中获取数据,第7行如果 `overrideValueNode` 不为空,则第10~11行开始创建表达式,形如 `#{sharenui}`,通过这里的处理可知任务节点中定义的节点缓存可以使用 Juel 表达式。

8.12 动态修改任务节点缓存数据

接来自定义一个类,该类主要用于生成和修改节点缓存数据,进而验证任务节点缓存功能是否可以正常使用,该类的相关定义如代码清单 8-21 所示。

代码清单 8-21 JedisTest.java

```

1 public static void main(String[] args) {
2     Jedis jedis = new Jedis("127.0.0.1");           //Redis 的 ip 端口
3     //自定义 VO 对象
4     ProcessDefinitionInfoCacheObjectVo v = new ProcessDefinitionInfoCacheObjectVo();
5     String string = "{\"bpmn\":{\"usertask1\":{\"userTaskName\":\"#{shaneui}\",
6         \"userTaskAssignee\":\"shaneui\"}}}\";
7     v.setInfoNode(string);v.setRevision(1);
8     //流程定义 id
9     String key = "sharenuiNodeCache:operationProcess:1:4";
10    //添加到 redis 中 其中 key 值与自定义缓存处理类中 get 方法的 key 一致
11    jedis.set(key.getBytes(), MyProcessDefinitionInfoCache.toByteArray(v));
12 }

```

上述准备工作完毕,接着部署代码清单 8-6 中定义的流程文档,其中新生成的流程定义 id 值为 `operationProcess:1:4`,然后执行上述代码修改任务节点缓存数据,最后启动流程实例以观察数据库中 `ACT_RU_TASK` 表的数据变化,启动流程实例相关实现如代码清单 8-22 所示。

代码清单 8-22 App.java

```

1 @Test
2 public void startProcessInstanceById() {
3     Map<String, Object> map = new HashMap<String, Object>();

```

```
4 map.put("shaneiu", "shaneiu1");
5 runtimeService.startProcessInstanceById("operationProcess:1:4", map);
6 map.clear();
7 map.put("shaneiu", "shaneiu2");
8 runtimeService.startProcessInstanceById("operationProcess:1:4", map);
9 }
```

为了方便测试,直接将需要的变量以及值添加到第3行定义的 Map 集合中,启动流程实例之后第6行将 Map 集合中的数据清空,然后再次设置新的值,第8行重新启动流程实例,ACT_RU_TASK 表的数据变化如图 8-7 所示。

ID_	REV_	PROC_DEF_ID_	NAME_
5005	1	operationProcess:1:4	shaneiu1
5011	1	operationProcess:1:4	shaneiu2

图 8-7 ACT_RU_TASK 表的数据变化

通过图 8-7 可知修改任务节点缓存功能已经生效。

8.13 节点缓存使用技巧

授之以鱼不如授之以渔,在这里教一个可以迅速查看哪些节点支持节点缓存功能的小技巧。首先,打开 eclipse 等开发工具定位到 ProcessEngineConfiguration 类,然后查找该类中的 isEnabledProcessDefinitionInfoCache 方法单击右键,选 Reference> Workspace,这样所有可以使用节点缓存特性的类都会被查询出来,效果如图 8-8 所示。

```
▼ org.activiti.engine.impl.bpmn.behavior
  > ScriptTaskActivityBehavior
  > ServiceTaskDelegateExpressionActivityBehavior
  > ServiceTaskExpressionActivityBehavior
  > UserTaskActivityBehavior
  > org.activiti.engine.impl.bpmn.deployer
  > org.activiti.engine.impl.bpmn.helper
  > org.activiti.engine.impl.el
  > org.activiti.engine.impl.scripting
```

图 8-8 支持节点缓存功能的类

通过图 8-8 可知节点缓存功能目前只运用于部分节点,如果使用的节点不支持节点缓存功能,可以使用扩展源码的方式进行改造。

第 9 章

定时作业

对于作业执行器而言,Activiti 分为作业执行器和异步作业执行器,两者设计的目的均是为了处理定时作业,需要了解一点,Activiti 的 5.17.0 版本中引入了异步作业执行器,相比作业执行器,异步作业执行器配置性更强,扩展性更好,但由于是 Activiti 提供的新特性,没有经过工业环境太多的实践验证,所以本章重点讲解作业执行器的实现过程。作业执行器和异步作业执行器的实现原理类似,关于异步作业执行器可以结合本章的讲解以及 ProcessEngineConfigurationImpl 类中的 initAsyncExecutor 方法自行学习。

9.1 初始化作业执行器

工欲善其事必先利其器,首先分析 initJobExecutor 方法的实现逻辑,该方法位于 ProcessEngineConfigurationImpl 类中,主要用来初始化作业执行器,相关实现如代码清单 9-1 所示。

代码清单 9-1 ProcessEngineConfigurationImpl.java

```
1 protected void initJobExecutor() {
2     if (isAsyncExecutorEnabled() == false) {
3         if (jobExecutor == null) {
4             jobExecutor = new DefaultJobExecutor();
5         }
6         jobExecutor.setClockReader(this.clock);
7         jobExecutor.setCommandExecutor(commandExecutor);
8         jobExecutor.setAutoActivate(jobExecutorActivate);
9         if (jobExecutor.getRejectedJobsHandler() == null) {
10             if (customRejectedJobsHandler != null) {
11                 jobExecutor.setRejectedJobsHandler(customRejectedJobsHandler);
12             } else {
13                 jobExecutor.setRejectedJobsHandler(new CallerRunsRejectedJobsHandler());
14             }
15         }
16     }
17 }
```

```

15     }
16 }
17 }

```

下面概括 initJobExecutor 方法的处理逻辑。

(1) 上述代码中,第 2 行调用 isAsyncExecutorEnabled 方法进行处理,该方法的实现如代码清单 9-2 所示。

代码清单 9-2 ProcessEngineConfigurationImpl.java

```

1 protected boolean asyncExecutorEnabled;
2 public boolean isAsyncExecutorEnabled() {
3     return asyncExecutorEnabled;
4 }

```

其中,asyncExecutorEnabled 为 ProcessEngineConfigurationImpl 类中的开关属性,表示是否开启异步作业执行器功能,如果该开关属性值为 false,则初始化作业执行器,通过此方法的处理可以看出,对于开发人员来说作业执行器和异步作业执行器功能只能选用一个。

(2) 代码清单 9-1 中,第 3 行判断 jobExecutor 开关属性值是否为空,如果该属性值为空,第 4 行开始实例化 DefaultJobExecutor 类(默认作业执行器)。

(3) 代码清单 9-1 中,第 6~8 行为 jobExecutor 对象填充如下属性值:① clock;② commandExecutor(命令执行器);③ jobExecutorActivate(作业执行器是否激活)。只有 jobExecutorActivate 开关属性值为 true,引擎才会开启作业执行器功能,使用时要特别注意。

(4) 代码清单 9-1 中,第 9 行获取 jobExecutor 对象中的 getRejectedJobsHandler 方法返回值,如果该方法的返回值不为空,程序不予处理,否则第 10 行开始判断 customRejectedJobsHandler 开关属性值是否为空,如果该属性值不为空,则第 11 行代码将该属性值设置到 jobExecutor 对象的 rejectedJobsHandler 属性中;如果该属性值为空执行第 13 行代码。

注意

上述代码中涉及的重要开关属性有 asyncExecutorEnabled、jobExecutor、jobExecutorActivate、customRejectedJobsHandler。

9.2 初始化作业处理器

Activiti 将一系列的作业按照类型进行归类,这样同种类型的作业处理器均为同一个,也方便程序的维护。下面讲解作业处理器的初始化过程,initJobHandlers 方法用于初始化作业处理器,该方法位于 ProcessEngineConfigurationImpl 类中,具体实现如代码清单 9-3 所示。

代码清单 9-3 ProcessEngineConfigurationImpl.java

```

1 protected void initJobHandlers() {
2     jobHandlers = new HashMap<String, JobHandler>();
3     TimerExecuteNestedActivityJobHandler timerExecuteNestedActivityJobHandler = new
4     TimerExecuteNestedActivityJobHandler();

```

```

5     jobHandlers.put(timerExecuteNestedActivityJobHandler.getType(),
6         timerExecuteNestedActivityJobHandler);
7     TimerCatchIntermediateEventJobHandler timerCatchIntermediateEvent = new
8         TimerCatchIntermediateEventJobHandler();
9     jobHandlers.put(timerCatchIntermediateEvent.getType(), timerCatchIntermediateEvent);
10    TimerStartEventJobHandler timerStartEvent = new TimerStartEventJobHandler();
11    jobHandlers.put(timerStartEvent.getType(), timerStartEvent);
12    AsyncContinuationJobHandler asyncContinuationJobHandler = new AsyncContinuationJobHandler();
13    jobHandlers.put(asyncContinuationJobHandler.getType(), asyncContinuationJobHandler);
14    TimerSuspendProcessDefinitionHandler suspendProcessDefinitionHandler = new
15        TimerSuspendProcessDefinitionHandler();
16    jobHandlers.put(suspendProcessDefinitionHandler.getType(),
17        suspendProcessDefinitionHandler);
18    TimerActivateProcessDefinitionHandler activateProcessDefinitionHandler = new
19        TimerActivateProcessDefinitionHandler();
20    jobHandlers.put(activateProcessDefinitionHandler.getType(),
21        activateProcessDefinitionHandler);
22    if (getCustomJobHandlers() != null) {
23        for (JobHandler customJobHandler : getCustomJobHandlers()) {
24            jobHandlers.put(customJobHandler.getType(), customJobHandler);
25        }
26    }
27 }

```

下面概括上述代码的处理逻辑。

(1) 第 2 行初始化 jobHandlers 集合,通过程序在这里的处理可以看出,引擎并没有将其设置为开关属性,而是直接初始化了该集合。

(2) 第 3~6 行实例化 TimerExecuteNestedActivityJobHandler 类(任务超时作业处理器),并将其添加到 jobHandlers 集合中。

(3) 第 7~9 行实例化 TimerCatchIntermediateEventJobHandler 类(定时任务作业处理器),并将其添加到 jobHandlers 集合中。

(4) 第 10~11 行实例化 TimerStartEventJobHandler 类(定时启动流程实例作业处理器),并将其添加到 jobHandlers 集合中。

(5) 第 12~13 行实例化 AsyncContinuationJobHandler 类(异步节点处理器),并将其添加到 jobHandlers 集合中。

(6) 第 14~17 行实例化 TimerSuspendProcessDefinitionHandler 类(挂起流程定义处理器),并将其添加到 jobHandlers 集合中。

(7) 第 18~21 行实例化 TimerActivateProcessDefinitionHandler 类(激活流程定义处理器),并将其添加到 jobHandlers 集合中。

(8) 第 22 行获取 getCustomJobHandlers 方法的返回值,该方法的实现如代码清单 9-4 所示。

getCustomJobHandlers 方法用于获取 customJobHandlers 集合,customJobHandlers 为开关属性,开发人员可以对其进行配置使用。

(9) 第 23~25 行循环遍历 getCustomJobHandlers 方法的返回值,并将遍历之后结果添加到 jobHandlers 集合中。

代码清单 9-4 ProcessEngineConfigurationImpl.java

```
1 protected List<JobHandler> customJobHandlers;  
2 public List<JobHandler> getCustomJobHandlers() {  
3     return customJobHandlers;  
4 }
```

扩展

可以通过设置 customJobHandlers 开关属性替换引擎默认的作业处理器。

接下来,详细讲解上述涉及的作业处理器。

9.2.1 任务超时作业

TimerExecuteNestedActivityJobHandler 为任务超时作业处理器。在实际项目开发中,可以为任务节点绑定一个定时边界事件,如果任务节点在指定的时间之内没有结束,则流程实例会按照边界事件的流向进行运转,接下来定义一个流程文档如代码清单 9-5 所示,该流程文档对应的流程图如图 9-1 所示。

代码清单 9-5 timer-transition.bpmn

```
1 <process id="timer-transition" name="timer-transition" isExecutable="true">  
2     <startEvent id="startevent1" name="Start"></startEvent>  
3     <userTask id="usertask1" name="任务 1"></userTask>  
4     <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>  
5     <userTask id="usertask2" name="任务 2"></userTask>  
6     <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"></sequenceFlow>  
7     <boundaryEvent id="boundarytimer1" name="Timer" attachedToRef="usertask1"  
8         cancelActivity="true">  
9         <timerEventDefinition>  
10             <timeDuration>PT60S</timeDuration>  
11         </timerEventDefinition>  
12     </boundaryEvent>  
13     <endEvent id="endevent1" name="End"></endEvent>  
14     <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>  
15     <userTask id="usertask3" name="任务 3"></userTask>  
16     <sequenceFlow id="flow4" sourceRef="boundarytimer1" targetRef="usertask3">  
17 </sequenceFlow>  
18     <endEvent id="endevent2" name="End"></endEvent>  
19     <sequenceFlow id="flow5" sourceRef="usertask3" targetRef="endevent2"></sequenceFlow>  
20 </process>
```

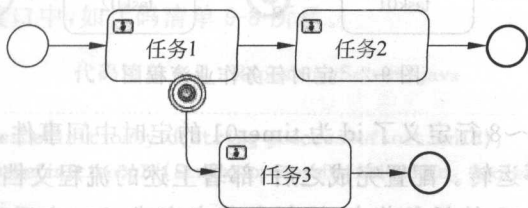


图 9-1 任务超时作业流程图

在上述代码中,第7~12行为id为usertask1的任务节点定义了一个id为boundarytimer1的边界事件,其中第10行定义了该任务节点的超时时间为60s,第8行的cancelActivity属性说明如下。

(1) cancelActivity 属性为 true: 当作业被处理时,当前的活动节点(对应第7行中的attachedToRef 属性值)会被中断,可以称之为中断事件,引擎默认使用的是中断事件方式。

(2) cancelActivity 属性为 false: 当作业被处理时,当前的活动节点不会被中断,即当前的活动节点会继续存活,例如上述代码中id为usertask1的任务节点会保留于ACT_RU_TASK表中,但是引擎会创建一个新的分支,从而可以让流程实例沿着id为boundarytimer1节点对应的流向继续向下运转,可以称之为非中断事件。

扩展

cancelActivity 处理过程可以参考 BoundaryEventActivityBehavior 类的 execute 方法。

9.2.2 定时任务作业

TimerCatchIntermediateEventJobHandler 为定时任务作业处理器。在实际的项目开发中,可以使用定时中间事件来指定流程实例到达下一个任务节点的时间。定义一个流程文档如代码清单 9-6 所示,该流程文档对应的流程图如图 9-2 所示。

代码清单 9-6 timer-intermediate-transition.xml

```

1 <process id="timerIntermediateEventTest01" isExecutable="true">
2   <startEvent id="start"></startEvent>
3   <userTask id="task01" name="task01"></userTask>
4   <intermediateCatchEvent id="timer01">
5     <timerEventDefinition>
6       <timeDuration>PT20S</timeDuration>
7     </timerEventDefinition>
8   </intermediateCatchEvent>
9   <userTask id="task02" name="task02"></userTask>
10  <endEvent id="end"></endEvent>
11  <sequenceFlow id="f2" sourceRef="task01" targetRef="timer01"></sequenceFlow>
12  <sequenceFlow id="f3" sourceRef="timer01" targetRef="task02"></sequenceFlow>
13  <sequenceFlow id="f4" sourceRef="task02" targetRef="end"></sequenceFlow>
14  <sequenceFlow id="f1" sourceRef="start" targetRef="task01"></sequenceFlow>
15 </process>

```

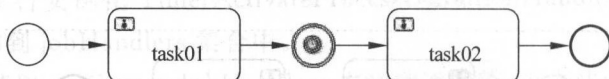


图 9-2 定时任务作业流程图

在上述代码中,第4~8行定义了id为timer01的定时中间事件,其中第6行指定了20s之后,流程实例继续向下运转。配置完成之后,部署上述的流程文档并启动流程实例,流程实例首先到达id为task01的任务节点,该任务节点完成,20s之后,流程实例会到达id为

task02 的任务节点。如果去除第 4~8 行代码,则上述同样的操作,完成 id 为 task01 的任务节点之后流程实例会立刻到达 id 为 task02 的任务节点。

扩展

intermediateCatchEvent 的行为类为 IntermediateCatchEventActivityBehavior。

9.2.3 定时启动流程实例作业

TimerStartEventJobHandler 为定时启动流程实例作业处理器,在实际项目开发中,可以在定时开始节点中配置启动该流程实例的时间阈值,这样当流程文档部署完毕,超过时间阈值后流程实例会被作业处理器启动。接下来定义一个流程文档如代码清单 9-7 所示,该流程文档对应的流程图如图 9-3 所示。

代码清单 9-7 timer-start-event.bpmn

```
1 <process id="myProcess" name="timer-start-event" isExecutable="true">
2   <startEvent id="timerstartevent1" name="Timer start">
3     <timerEventDefinition>
4       <timeDuration>PT20S</timeDuration>
5     </timerEventDefinition>
6   </startEvent>
7   <userTask id="usertask1" name="User Task"></userTask>
8   <sequenceFlow id="flow1" sourceRef="timerstartevent1" targetRef="usertask1">
</sequenceFlow>
9   <endEvent id="endevent1" name="End"></endEvent>
10  <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="endevent1"></sequenceFlow>
11 </process>
```

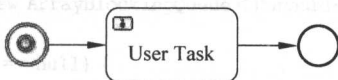


图 9-3 定时启动流程实例作业流程图

在上述代码中,第 2~5 行定义了 id 为 timerstartevent1 的定时开始节点,其中第 4 行配置了 20s 之后启动该流程文档所对应的流程实例。

9.2.4 其他作业

Activiti 在 5.11 版本中增加了挂起流程的特性,挂起流程与激活流程的方法均定义在 RepositoryService 接口中,如代码清单 9-8 所示。

代码清单 9-8 RepositoryService.java

```
1 void suspendProcessDefinitionById(String processDefinitionId);
2 void suspendProcessDefinitionById(String processDefinitionId, boolean suspendProcessInstances,
Date suspensionDate);
3 void activateProcessDefinitionById(String processDefinitionId);
```

```
4 void activateProcessDefinitionById(String processDefinitionId, boolean activateProcessInstances,
    Date activationDate);
```

在上述代码中,第 1~2 行为挂起流程的方法定义,第 3~4 行为激活流程的方法定义,其中第 2 行中三个参数的含义如下。

(1) processDefinitionId: 流程定义 id。

(2) suspendProcessInstances: 如果该参数为 true,则该流程对应的所有流程实例会被挂起。

(3) suspensionDate: 流程挂起的时间,如果该参数值为空,则流程立刻被挂起,否则将其添加到作业中。

注意

如果流程实例被挂起,那么存活于 ACT_RU_TASK 表中的任务节点是不能继续完成的,如果程序强行完成这些存活的任务,则报错,关于这一点可以参考 NeedsActiveTaskCmd 类中的 execute 方法。

以上作业处理器的执行过程可以参考第 14 章。

9.3 作业执行器原理

第 2.6 节中涉及了作业执行器的启动,形如 jobExecutor.start(),该方法的实现如代码清单 9-9 所示。

代码清单 9-9 JobExecutor.java

```
1 protected boolean isActive = false;
2 public void start() {
3     if (isActive) {
4         return;
5     }
6     ensureInitialization();
7     startExecutingJobs();
8     isActive = true;
9 }
```

将上述代码的处理逻辑总结如下。

(1) 第 3 行判断 isActive 属性值是否为 true,该属性值默认为 false,如果为 true,表示作业执行器已经启动成功,因此该方法直接返回;否则开始执行如下的逻辑。

(2) 第 6 行调用 ensureInitialization 方法确保必要的类已经被实例化。

(3) 第 7 行调用 startExecutingJobs 方法启动作业执行器。

(4) 第 8 行设置 isActive 属性值为 true。

9.3.1 初始化作业执行器

ensureInitialization 方法的实现如代码清单 9-10 所示。

代码清单 9-10 JobExecutor.java

```
1 protected Command<AcquiredJobs> acquireJobsCmd;  
2 protected AcquireJobsRunnable acquireJobsRunnable;  
3 protected void ensureInitialization() {  
4     if (acquireJobsCmd == null) {  
5         acquireJobsCmd = new AcquireJobsCmd(this);  
6     }  
7     if (acquireJobsRunnable == null) {  
8         acquireJobsRunnable = new AcquireJobsRunnableImpl(this);  
9     }  
10 }
```

在上述代码中,如果 `acquireJobsCmd` 为空,则执行第 5 行实例化 `AcquireJobsCmd` 类,其中 `this` 为 `DefaultJobExecutor` 实例对象(前提是开发人员使用的是默认作业执行器)。如果 `acquireJobsRunnable` 为空,则执行第 8 行实例化 `AcquireJobsRunnableImpl` 类,该类实现了 `AcquireJobsRunnable` 接口,`AcquireJobsRunnable` 接口继承 `Runnable` 接口,因此该类是一个非常重要的线程类,负责定时查询并执行作业。

9.3.2 启动作业执行器

`startExecutingJobs` 方法的具体实现如代码清单 9-11 所示。

代码清单 9-11 DefaultJobExecutor.java

```
1 protected BlockingQueue<Runnable> threadPoolQueue;  
2 protected ThreadPoolExecutor threadPoolExecutor;  
3 protected void startExecutingJobs() {  
4     if (threadPoolQueue == null) {  
5         threadPoolQueue = new ArrayBlockingQueue<Runnable>(queueSize);  
6     }  
7     if (threadPoolExecutor == null) {  
8         threadPoolExecutor = new ThreadPoolExecutor(corePoolSize, maxPoolSize, keepAliveTime,  
9             TimeUnit.MILLISECONDS, threadPoolQueue);  
10        threadPoolExecutor.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());  
11    }  
12    startJobAcquisitionThread();
```

在上述代码中,如果 `threadPoolQueue` 为空,则执行第 5 行实例化 `ArrayBlockingQueue` 类。第 7 行如果 `threadPoolExecutor` 为空,则执行第 8 行实例化 `ThreadPoolExecutor`(线程池)类,第 9 行设置线程池队列饱和策略为 `AbortPolicy`,第 11 行调用 `startJobAcquisitionThread` 方法进行处理,该方法的具体实现如代码清单 9-12 所示。

代码清单 9-12 DefaultJobExecutor.java

```
1 protected Thread jobAcquisitionThread;  
2 protected AcquireJobsRunnable acquireJobsRunnable;  
3 protected void startJobAcquisitionThread() {
```



```

4     if (jobAcquisitionThread == null) {
5         jobAcquisitionThread = new Thread(acquireJobsRunnable);
6     }
7     jobAcquisitionThread.start();
8 }

```

在上述代码中,第 4 行判断 jobAcquisitionThread 属性值是否为空,该属性为 Thread 类型,如果该属性为空,则执行第 5 行实例化 Thread 类,并传入 acquireJobsRunnable 参数,第 7 行启动 jobAcquisitionThread 线程。

9.4 添加定时作业

看到上文一系列的讲解,可能会有这样的疑问:作业是如何被记录到数据库并执行呢?首先分析作业的添加过程,本节以定时启动流程实例作业为例进行讲解。7.3.7 节提到了 scheduleTimers(timers) 方法,该方法的具体实现如代码清单 9-13 所示。

代码清单 9-13 BpmnDeployer.java

```

1 protected void scheduleTimers(List<TimerEntity> timers) {
2     for (TimerEntity timer : timers) {
3         Context.getCommandContext().getJobEntityManager().schedule(timer);
4     }
5 }

```

在上述代码中,第 2 行循环遍历 timers 集合,第 3 行调用 JobEntityManager 类中的 schedule 方法,schedule 方法的相关实现如代码清单 9-14 所示。

代码清单 9-14 JobEntityManager.java

```

1 public void schedule(TimerEntity timer) {
2     Date dueDate = timer.getDueDate();
3     if (dueDate == null) {
4         throw new ActivitiIllegalArgumentException("dueDate is null");
5     }
6     timer.insert();
7     ProcessEngineConfiguration engineConfiguration = Context.getProcessEngineConfiguration();
8     if (engineConfiguration.isAsyncExecutorEnabled() == false && timer.getDueDate().
    getTime() <=
9         (engineConfiguration.getClock().getCurrentTime().getTime())) {
10         hintJobExecutor(timer);
11     }
12 }

```

在上述代码中,第 2 行获取 dueDate(作业到期的时间),如果该值为空,则第 4 行程序直接报错,试想一下,如果没有明确指定作业的到期时间,该作业什么时候完成呢?第 6 行将 timer 添加到会话缓存中,最终该会话缓存的数据会刷新到 ACT_RU_JOB 表中。

第 7 行获取 ProcessEngineConfiguration 实例对象,第 8 行中,如果 engineConfiguration 对象中的 asyncExecutorEnabled 开关属性值为 false,并且作业的到期时间小于或者等于当前系统时间,则执行第 10 行 hintJobExecutor 方法。例如,当前系统时间为"2016-12-23 13:34:09",但是作业到期时间为"2016-10-23",那就说明当前的作业可以执行。hintJobExecutor 方法的相关实现如代码清单 9-15 所示。

代码清单 9-15 JobEntityManager.java

```
1 protected void hintJobExecutor(JobEntity job) {
2     JobExecutor jobExecutor = Context.getProcessEngineConfiguration().getJobExecutor();
3     TransactionListener transactionListener = new JobAddedNotification(jobExecutor);
4     Context.getCommandContext().getTransactionContext().addTransactionListener(
5         TransactionState.COMMITTED, transactionListener);
6 }
```

在上述代码中,第 2 行获取作业执行器 JobExecutor 实例对象,第 3 行实例化 TransactionListener 类,第 4~5 行代码操作的意图可以参考第 12.8.5 节。其中,JobAddedNotification 类的核心定义如代码清单 9-16 所示。

代码清单 9-16 JobAddedNotification.java

```
1 public class JobAddedNotification implements TransactionListener {
2     protected JobExecutor jobExecutor;
3     public JobAddedNotification(JobExecutor jobExecutor) {
4         this.jobExecutor = jobExecutor;
5     }
6     public void execute(CommandContext commandContext) {
7         jobExecutor.jobWasAdded();
8     }
9 }
```

接下来,分析第 7 行代码 jobExecutor 对象中 jobWasAdded() 方法所实现的功能,该方法的相关实现如代码清单 9-17 所示。

代码清单 9-17 JobExecutor.java

```
1 protected boolean isActive = false;
2 public void jobWasAdded() {
3     if(isActive){
4         acquireJobsRunnable.jobWasAdded();
5     }
6 }
```

在上述代码中,如果 isActive 属性值为 true,则执行第 4 行代码,调用 acquireJobsRunnable 对象中的 jobWasAdded 方法,该方法的实现如代码清单 9-18 所示。

代码清单 9-18 AcquireJobsRunnableImpl.java

```

1  protected volatile boolean isJobAdded = false;
2  protected final Object MONITOR = new Object();
3  protected final AtomicBoolean isWaiting = new AtomicBoolean(false);
4  public void jobWasAdded() {
5      isJobAdded = true;
6      if(isWaiting.compareAndSet(true, false)) {
7          synchronized (MONITOR) {
8              MONITOR.notifyAll();
9          }
10     }
11 }

```

在上述代码中,第 5 行设置 isJobAdded 为 true,表示已经有需要执行的作业了, isWaiting 表示当前线程也就是 AcquireJobsRunnableImpl 类是否处于等待执行作业状态,如果该值为 true,则执行第 8 行代码,MONITOR.notifyAll() 被调用之后,JVM 可以选择任何一个调用了 MONITOR.wait() 的线程投入运行,那么如何知道哪些线程调用了 MONITOR.wait()? 这也是接下来需要重点讲解的地方。

9.5 执行定时作业

接下来,分析 AcquireJobsRunnableImpl 类中的 run 方法的处理逻辑,该方法的相关实现如代码清单 9-19 所示。

代码清单 9-19 AcquireJobsRunnableImpl.java

```

1  public synchronized void run() {
2      final CommandExecutor commandExecutor = jobExecutor.getCommandExecutor();
3      while (!isInterrupted) {
4          isJobAdded = false;
5          int maxJobsPerAcquisition = jobExecutor.getMaxJobsPerAcquisition();
6          try {
7              AcquiredJobs acquiredJobs = commandExecutor.execute(jobExecutor.getAcquireJobsCmd());
8              for (List<String> jobIds : acquiredJobs.getJobIdBatches()) {
9                  jobExecutor.executeJobs(jobIds);
10             }
11             millisToWait = jobExecutor.getWaitTimeInMillis();
12             int jobsAcquired = acquiredJobs.getJobIdBatches().size();
13             if (jobsAcquired >= maxJobsPerAcquisition) {
14                 millisToWait = 0;
15             }
16         } catch (Throwable e) {
17             millisToWait = jobExecutor.getWaitTimeInMillis();
18         }
19         if ((millisToWait > 0) && (!isJobAdded)) {
20             try {

```

```

22 synchronized (MONITOR) {
23     if(!isInterrupted) {
24         isWaiting.set(true);
25         MONITOR.wait(millisToWait);
26     }
27 }
28 }finally {
29     isWaiting.set(false);
30 }
31 }
32 }
33 }

```

AcquireJobsRunnableImpl 类为线程类,因此只要 isInterrupted 为 true,则第 3~33 行代码块会一直执行,关于这一点需要牢记于心。将上述代码的处理逻辑梳理总结如下。

(1) 第 2 行获取命令执行器对象。

(2) 第 5 行获取当前作业执行器可以完成的作业个数,该值默认为 1。

(3) 第 7 行执行 AcquireJobsCmd 命令,并使用 acquiredJobs 变量存储执行命令之后的返回结果。

(4) 第 8~9 行循环遍历 acquiredJobs 的 getJobIdBatches() 方法的返回值,并依此调用 jobExecutor 的 executeJobs 方法处理作业。

(5) 第 11 行获取当前线程需要等待的时间,该时间默认为 5s。

(6) 第 12 行获取当前作业执行器需要处理的作业个数。

(7) 第 13~14 行,如果 jobsAcquired 大于或者等于 maxJobsPerAcquisition,那么就需设置 millisToWait 为 0,该操作是为了确保需要处理的作业都可以被处理。

(8) 上述一系列操作,任何一个环节出错,则执行第 18 行代码。

(9) 第 20~30 行,如果 millisToWait 大于 0,并且 isJobAdded 为 false,则需要让当前的线程等待一段时间,当前线程需要等待的时间为 millisToWait 值。

AcquireJobsCmd 类的核心定义如代码清单 9-20 所示。

代码清单 9-20 AcquireJobsCmd.java

```

1 public AcquiredJobs execute(CommandContext commandContext) {
2     String lockOwner = jobExecutor.getLockOwner();
3     int lockTimeInMillis = jobExecutor.getLockTimeInMillis();
4     int maxNonExclusiveJobsPerAcquisition = jobExecutor.getMaxJobsPerAcquisition();
5     AcquiredJobs acquiredJobs = new AcquiredJobs();
6     List<JobEntity> jobs = commandContext
7         .getJobEntityManager()
8         .findNextJobsToExecute(new Page(0, maxNonExclusiveJobsPerAcquisition));
9     for (JobEntity job: jobs) {
10         List<String> jobIds = new ArrayList<String>();
11         if (job != null && !acquiredJobs.contains(job.getId())) {
12             if (job instanceof MessageEntity && job.isExclusive() && job.getProcessInstanceId() != null) {
13                 try {

```



```

14         Thread.sleep(100);
15     } catch (InterruptedException e) {}
16     List<JobEntity> exclusiveJobs = commandContext.getJobEntityManager()
17         .findExclusiveJobsToExecute(job.getProcessInstanceId());
18     for (JobEntity exclusiveJob : exclusiveJobs) {
19         if (exclusiveJob != null) {
20             lockJob(commandContext, exclusiveJob, lockOwner, lockTimeInMillis);
21             jobIds.add(exclusiveJob.getId());
22         }
23     }
24     } else {
25         lockJob(commandContext, job, lockOwner, lockTimeInMillis);
26         jobIds.add(job.getId());
27     }
28     }
29     acquiredJobs.addJobIdBatch(jobIds);
30 }
31 return acquiredJobs;
32 }

```

对于上述代码的处理逻辑,可以将其梳理总结如下。

- (1) 第 3 行获取作业的锁定时间,默认为 5min。
- (2) 第 5 行实例化 AcquiredJobs 类,该类记录了所有可以执行的作业 id 值。
- (3) 第 6~8 行,从 ACT_RU_JOB 表中查询所有可以执行的作业,对应的 SQL 如代码清单 9-21 所示。

代码清单 9-21 Job.xml

```

1 SELECT RES. *
2 FROM ACT_RU_JOB RES
3 LEFT OUTER JOIN ACT_RU_EXECUTION PI ON PI.ID_ = RES.PROCESS_INSTANCE_ID_
4 WHERE (RES.RETRIES_ > 0)
5     AND (RES.DUEDATE_ IS NULL
6         OR RES.DUEDATE_ <= ?)
7     AND (RES.LOCK_OWNER_ IS NULL
8         OR RES.LOCK_EXP_TIME_ <= ?)
9     AND ((RES.EXECUTION_ID_ IS NULL)
10        OR (PI.SUSPENSION_STATE_ = 1)) LIMIT ?
11 OFFSET ?

```

该 SQL 语句以 DUEDATE_、LOCK_EXP_TIME_、LOCK_OWNER_ 字段作为检索条件从 ACT_RU_JOB 表中查询数据,其中第 6 行和第 8 行需要的参数值均为调用该 SQL 语句的当前系统时间。

(4) 第 9 行遍历 jobs 集合,第 11 行如果 job 不为空并且该作业不存在于 acquiredJobs 对象中的 acquiredJobs 集合中,则开始执行第 12~28 行代码;否则执行第 29 行代码。其中第 12 行首先判断 job 是否为 MessageEntity 实例对象,然后判断 job 是否是独占模式,最

后判断 job 对象中的 processInstanceId 属性值是否为空,如果以上三个判断都为 true,则开始执行第 14 行代码,即将当前线程休眠 100 毫秒,如果出现异常则执行第 16~21 行代码;否则执行第 25~26 行代码。看到这个地方的处理对独占模式的使用恍然大悟。

(5) 第 25 行调用 lockJob 方法,该方法主要是为 job 对象填充属性:①lockOwner(对应 ACT_RU_JOB 表中的 LOCK_OWNER_字段);②lockExpirationTime(对应 ACT_RU_JOB 表中的 LOCK_EXP_TIME_字段)。该操作与步骤 3 前呼后应。

(6) 第 29 行调用 acquiredJobs 对象的 addJobIdBatch 方法,该方法的相关实现如代码清单 9-22 所示。

代码清单 9-22 AcquiredJobs.java

```
1 public class AcquiredJobs {
2     protected List<List<String>> acquiredJobBatches = new ArrayList<List<String>>();
3     protected Set<String> acquiredJobs = new HashSet<String>();
4     public void addJobIdBatch(List<String> jobIds) {
5         acquiredJobBatches.add(jobIds);
6         acquiredJobs.addAll(jobIds);
7     }
8 }
```

AcquiredJobs 类主要用于记录需要处理的作业,其内部持有的 acquiredJobBatches 和 acquiredJobs 集合主要用来记录需要处理的作业 id 值。

9.6 处理作业

从 ACT_RU_JOB 表中查询作业的处理逻辑已经讲解完毕,接下来重点分析引擎是如何处理这些作业的。上文中 jobExecutor.executeJobs(jobIds) 方法主要用于处理作业,executeJobs 方法的相关实现如代码清单 9-23 所示。

代码清单 9-23 DefaultJobExecutor.java

```
1 public void executeJobs(List<String> jobIds) {
2     try {
3         threadPoolExecutor.execute(new ExecuteJobsRunnable(this, jobIds));
4     } catch (RejectedExecutionException e) {
5         rejectedJobsHandler.jobsRejected(this, jobIds);
6     }
7 }
```

在上述代码中,第 3 行开始执行 ExecuteJobsRunnable 类,该类实现了 Runnable 类,如果执行过程中出现了异常,则执行第 5 行代码(作业处理失败之后,再次尝试执行一次),接下来分析 ExecuteJobsRunnable 类,该类的定义如代码清单 9-24 所示。

代码清单 9-24 ExecuteJobsRunnable.java

```

1 public void run() {
2     if (jobIds != null) {
3         handleMultipleJobs();
4     }
5     if (job != null) {
6         handleSingleJob();
7     }
8 }

```

将上述代码的处理逻辑总结如下。

- (1) 如果 jobIds 不为空,则执行第 3 行调用 handleMultipleJobs 方法批量处理作业。
- (2) 如果 job 不为空,则执行第 6 行调用 handleSingleJob 方法处理单个作业。该过程可以参考批量处理作业的讲解。

9.6.1 批量处理作业

handleMultipleJobs 方法的具体实现代码清单 9-25 所示。

代码清单 9-25 ExecuteJobsRunnable.java

```

1 protected void handleMultipleJobs() {
2     final MultipleJobsExecutorContext jobExecutorContext = new MultipleJobsExecutorContext();
3     final List<String> currentProcessorJobQueue =
4     jobExecutorContext.getCurrentProcessorJobQueue();
5     final CommandExecutor commandExecutor = jobExecutor.getCommandExecutor();
6     currentProcessorJobQueue.addAll(jobIds);
7     Context.setJobExecutorContext(jobExecutorContext);
8     try {
9         while (!currentProcessorJobQueue.isEmpty()) {
10             String currentJobId = currentProcessorJobQueue.remove(0);
11             try {
12                 commandExecutor.execute(new ExecuteJobsCmd(currentJobId));
13             } catch (Throwable e) {
14             } finally {
15                 jobExecutor.jobDone(currentJobId);
16             }
17         }
18     } finally {
19         Context.removeJobExecutorContext();
20     }
21 }

```

在上述代码中,第 6 行将 jobIds 添加到 currentProcessorJobQueue 集合中,只要该集合不为空,则程序会执行第 9~17 行代码,其中第 10 行从 currentProcessorJobQueue 集合中移除一个作业,第 12 行调用命令执行器 commandExecutor 对象执行 ExecuteJobsCmd 命令,当作业执行完毕之后,第 15 行调用 jobExecutor 对象中的 jobDone 方法。接下来,重点

分析 ExecuteJobsCmd 类,该类的定义如代码清单 9-26 所示。

代码清单 9-26 ExecuteJobsCmd.java

```

1 public Object execute(CommandContext commandContext) {
2     if (jobId == null && job == null) {
3         throw new ActivitiIllegalArgumentException("jobId and job is null");
4     }
5     if (job == null) {
6         job = commandContext.getJobEntityManager().findJobById(jobId);
7     }
8     if (job == null) {
9         throw new JobNotFoundException(jobId);
10    }
11    JobExecutorContext jobExecutorContext = Context.getJobExecutorContext();
12    //获取作业处理上下文
13    if (jobExecutorContext != null) {
14        jobExecutorContext.setCurrentJob(job); //将 job 设置到 jobExecutorContext 对象中
15    }
16    FailedJobListener failedJobListener = null;
17    try {
18        failedJobListener = new
19        FailedJobListener(commandContext.getProcessEngineConfiguration().getCommandExecutor(),
20        jobId);
21        commandContext.getTransactionContext().addTransactionListener(TransactionState.
22        ROLLED_BACK, failedJobListener);
23        job.execute(commandContext);
24    } catch (Throwable exception) {
25        failedJobListener.setException(exception);
26        throw new ActivitiException("Job " + jobId + " failed", exception);
27    } finally {
28        if (jobExecutorContext != null) {
29            jobExecutorContext.setCurrentJob(null);
30        }
31    }
32    return null;
33 }

```

将上述代码的处理逻辑梳理总结如下。

(1) 第 2 行,如果 jobId 为空并且 job 为空,则程序报错。对于一个不存在的作业来说,执行上述代码显然没有意义。

(2) 第 5 行如果 job 为空,则执行第 6 行代码,从 ACT_RU_JOB 表中查询数据,如果没有查询到数据,则程序报错如第 9 行所示。通过程序在这里的处理可以看出作业中并没有使用缓存技术。

(3) 第 17~19 行实例化 FailedJobListener 类,该类可以在作业失败之后调用 JobRetryCmd 命令类再次重试执行作业,引擎默认重试三次。

(4) 第 22 行调用 job 对象的 execute 方法处理作业。因为 MessageEntity 类和 TimerEntity 类均继承 JobEntity 类,因此这里以 MessageEntity 类为例进行说明,该类中

execute 方法的相关实现如代码清单 9-27 所示。

代码清单 9-27 MessageEntity.java

```
1 public void execute(CommandContext commandContext) {
2     super.execute(commandContext);
3     delete();
4 }
```

上述代码的处理逻辑分为如下两个步骤。

- 第 2 行调用父类的 execute 方法,父类中该方法的实现如代码清单 9-28 所示。

代码清单 9-28 JobEntity.java

```
1 public void execute(CommandContext commandContext) {
2     ExecutionEntity execution = null;
3     if (executionId != null) {
4         execution = commandContext.getExecutionEntityManager().findExecutionById(executionId);
5     }
6     Map<String, JobHandler> jobHandlers =
7         Context.getProcessEngineConfiguration().getJobHandlers();
8     JobHandler jobHandler = jobHandlers.get(jobHandlerType);
9     jobHandler.execute(this, jobHandlerConfiguration, execution, commandContext);
10 }
```

在上述代码中,第 4 行通过 executionId 值获取 ExecutionEntity 实例对象,第 6~7 行获取所有的 jobHandlers,第 8 行根据 jobHandlerType 值从 jobHandlers 集合中获取该值对应的作业处理器,第 9 行调用 jobHandler 对象中的 execute 方法。

- 代码清单 9-27 第 3 行执行 delete 方法进行处理,该方法的实现如代码清单 9-29 所示。

代码清单 9-29 JobEntity.java

```
1 public void delete() {
2     Context.getCommandContext().getDbSqlSession().delete(this);
3     exceptionByteArrayRef.delete();
4     if (executionId != null) {
5         ExecutionEntity execution = Context.getCommandContext().getExecutionEntityManager()
6             .findExecutionById(executionId);
7         execution.removeJob(this);
8     }
9 }
```

在上述代码中,第 2 行从 ACT_RU_JOB 表中删除当前的作业数据,第 3 行从 ACT_GE_BYTEARRAY 表中删除该作业的信息,例如异常信息。第 5~6 行根据 executionId 获取 ExecutionEntity 实例对象,第 7 行的操作含义可以参考第 13.6.2 节。

(5) 第 27~28 行,如果作业处理完毕,从 `jobExecutorContext` 对象中移除该作业。

扩展

`FailedJobListener` 类的实现逻辑可以自行学习。

9.6.2 执行作业之异常处理

`rejectedJobsHandler.jobsRejected(this, jobIds)` 方法主要用于处理失败作业, `jobsRejected` 方法的实现如代码清单 9-30 所示。

代码清单 9-30 `CallerRunsRejectedJobsHandler.java`

```
1 public void jobsRejected(JobExecutor jobExecutor, List<String> jobIds) {  
2     try {  
3         new ExecuteJobsRunnable(jobExecutor, jobIds).run();  
4     } catch (Exception e) {  
5     }  
6 }
```

在上述代码中,第 3 行直接实例化 `ExecuteJobsRunnable` 类,并调用该类的 `run` 方法进行处理。

9.7 关闭作业执行器

第 2.7 节中涉及了作业执行器的关闭,形如 `jobExecutor.shutdown()`,该方法的具体实现如代码清单 9-31 所示。

代码清单 9-31 `JobExecutor.java`

```
1 public synchronized void shutdown() {  
2     if (!isActive) {  
3         return;  
4     }  
5     acquireJobsRunnable.stop();  
6     stopExecutingJobs();  
7     ensureCleanup();  
8     isActive = false;  
9 }
```

将上述代码的处理逻辑梳理总结如下。

(1) 第 2 行判断 `isActive` 属性值是否为 `true`,该值如果为 `false`,则该方法直接返回;否则开始执行如下的逻辑。

(2) 第 5 行调用 `acquireJobsRunnable` 的 `stop` 方法,该方法的相关实现如代码清单 9-32 所示。

代码清单 9-32 AcquireJobsRunnableImpl.java

```

1 public void stop() {
2     synchronized (MONITOR) {
3         isInterrupted = true;
4         if(isWaiting.compareAndSet(true, false)) {
5             MONITOR.notifyAll();
6         }
7     }
8 }

```

在上述代码中,第3行代码非常重要,只要 `isInterrupted` 为 `true`,则 `AcquireJobsRunnableImpl` 线程类会终止运行。

(3) 第6行调用 `stopExecutingJobs` 方法关闭作业执行器。

(4) 第7行调用 `ensureCleanup` 方法,将 `JobExecutor` 类中的 `acquireJobsCmd` 和 `acquireJobsRunnable` 设置为空。

(5) 第8行设置 `isActive` 属性值为 `false`。该操作与第2行代码的判断前呼后应。

9.8 自定义作业处理器

上文讲解了定时启动流程实例作业的处理程序 `TimerStartEventJobHandler`,要想使用该功能,就必须使用定时开始事件,而使用定时开始事件就需要大量的配置,例如配置开启时间、循环次数等。但是基于配置的方式不灵活,例如流程启动的时间需要通过程序自动计算,所以接下来考虑如何优雅的扩展作业处理器,该案例的实现可以细分为如下几个步骤。

(1) 自定义一个作业处理器,相关实现如代码清单 9-33 所示。

代码清单 9-33 ShareniumTimerStartEventJobHandler.java

```

1 public class ShareniumTimerStartEventJobHandler extends TimerStartEventJobHandler {
2     public static final String TYPE = "sharenium-timer-start-event";
3     public String getType() {
4         return TYPE;
5     }
6 }

```

在上述代码中,为了使自定义作业处理器与 `TimerStartEventJobHandler` 类有所区别,第2行代码定义了作业处理器的类型,并作为 `getType` 方法的返回值。

(2) 将自定义作业处理器注入流程引擎配置类如代码清单 9-34 所示。

代码清单 9-34 applicationContext.xml

```

1 <bean id="processEngineConfiguration"
2     class="org.activiti.spring.SpringProcessEngineConfiguration">
3     <property name="customJobHandlers">
4         <list>
5             <bean class="com.sharenium.activiti.learing.ch14.job.ShareniumTimerStartEventJobHandler"/>

```

```

6         </list>
7     </property>
8     <property name="jobExecutorActivate" value="true"></property>
9 </bean>

```

在上述代码中,第3~7行通过设置 customJobHandlers 开关属性将自定义的 Shareniu-TimerStartEventJobHandler 类注入流程引擎配置类。

(3) 自定义定时启动流程实例命令类,该类的定义如代码清单 9-35 所示。

代码清单 9-35 AutoStartProcessCmd.java

```

1 public class AutoStartProcessCmd implements Command<Void>{
2     private Date target; //开启流程实例的时间
3     private String processKey; //启动的流程 key
4     public AutoStartProcessCmd(Date target, String processKey) {
5         this.target = target;
6         this.processKey = processKey;
7     }
8     public Void execute(CommandContext commandContext) {
9         TimerEntity timer = new TimerEntity();
10        timer.setDueDate(target);
11        timer.setExclusive(true);
12        timer.setJobHandlerConfiguration(processKey);
13        timer.setJobHandlerType(ShareniuTimerStartEventJobHandler.TYPE);
14        Context.getCommandContext().getJobEntityManager().schedule(timer);
15        return null;
16    }
17 }

```

在上述代码中,第9行实例化 TimerEntity 类,第10~12行为 timer 对象填充属性,其中 target 为流程实例启动的时间,processKey 为流程的 key,第14行调度作业处理器。

(4) 部署流程文档,该流程文档的内容如代码清单 9-36 所示。

代码清单 9-36 shatenuauto.bpmn

```

1 <process id="shareniu" name="shareniu" isExecutable="true">
2     <startEvent id="startevent1" name="Start"></startEvent>
3     <userTask id="usertask1" name="User Task"></userTask>
4     <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
5     <endEvent id="endevent1" name="End"></endEvent>
6     <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="endevent1"></sequenceFlow>
7 </process>

```

(5) 步骤(4)执行完毕,接下来执行步骤(3)中定义的命令类,该过程如代码清单 9-37 所示。

代码清单 9-37 App.java

```

1 @Test
2 public void autoStartProcessCmd() throws Exception{

```



```

3      Date now = new Date();                                //获取系统时间
4      ServiceImpl serviceImpl = (ServiceImpl) repositoryService;
                                                    //获取 ServiceImpl 实例对象
5      serviceImpl.getCommandExecutor().execute(new AutoStartProcessCmd(now, "shareniu"));
6      Thread.sleep(1000);
7  }

```

在上述代码中,第 5 行执行 AutoStartProcessCmd 类,其中"shareniu"与步骤(4)中的 id="shareniu"一致,为了更快看出效果,可以先将第 6 行代码注释掉,然后执行上述代码,如不出意外,ACT_RU_JOB 表产生的数据如图 9-4 所示。

ID_	REV_	TYPE_	HANDLER_TYPE_	DUEDATE
60001	1	timer	shareniu-timer-start-event	2016-12-23 19:27:04.269

图 9-4 ACT_RU_JOB 表新增的作业

再次执行上述代码包含第 6 行代码,如不出意外,ACT_RU_TASK 表产生的数据如图 9-5 所示。

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_
62505	1	62502	62502	shareniu:1:17504
62509	1	62506	62506	shareniu:1:17504

图 9-5 ACT_RU_TASK 表新增的数据

第10章

流程虚拟机

第4章讲解了流程文档的解析原理,并实现了自定义元素属性的解析以及获取工作。流程文档中的元素解析完毕之后,是不可以直接使用的,还需要对其属性承载类实例(解析结果)进行一次加工,因为流程文档中定义的事件、网关、活动(流程三大要素)等信息最终交给流程虚拟机管理,而流程虚拟机的运转机制仅依赖 `ActivityImpl` 实例对象或者 `TransitionImpl` 实例对象(连线),并非元素解析后的 `BaseElement` 实例对象。试想一下,如果现在需要完成一个任务,对于流程虚拟机来说需要知道如下几个信息:该任务节点归属的流程文档(模板),任务完成之后流程实例运行的下个节点,也就是最终目的地等,如果流程虚拟机没有持有流程文档中所有元素的解析结果,可能流程虚拟机都不知道如何进行下一步操作,而且对于客户端来说流程虚拟机是一个完全不可触及的黑匣子,深入了解流程虚拟机的内部实现机制,有助于开发者掌握如何提高流程实例的运行效率,从而更好地设计流程文档。

接下来,本章重点讲解流程实例运转过程中是如何运用元素的解析结果,例如默认元素、自定义元素等。

10.1 流程虚拟机原理

在学习 `BaseElement` 实例对象的解析之前,首先需要从全局角度了解流程虚拟机到底做了什么工作。在流程文档元素解析并转化为 `Activiti` 中的内部表示 `BaseElement` 实例的过程中,深入讲解了 `Activiti` 使用开闭原则将流程文档中的元素与元素解析器一一对应,分离元素解析器的职责,使每一个元素解析器的职责更加单一,解析器之间不会相互影响,同样开闭原则在对象解析器这里也体现的淋漓尽致,在这里需要强调一点,本章讲解的解析工作主要是将 `BaseElement` 实例对象转换为流程虚拟机 `ActivityImpl` 实例对象或者 `TransitionImpl` 实例对象的过程,也许将该过程称之为对象解析可能会更加容易理解,该过程涉及的知识点比较复杂,首先使用图 10-1 进行通俗易懂的描述。

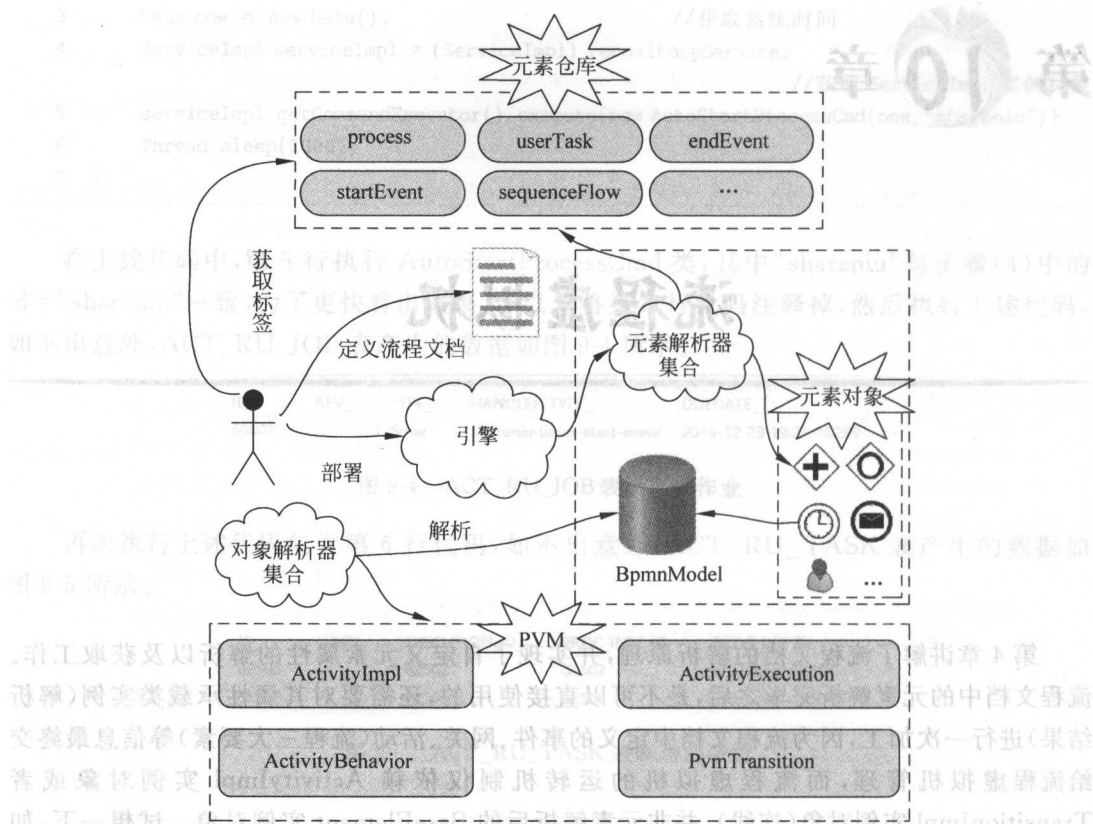


图 10-1 流程模型解析为流程虚拟机对象

将上图中每个模块的职责进行总结说明。

(1) 定义流程文档。

流程文档的定义工作完全交给客户端，客户端可以根据自己的业务需求定制流程文档，在流程文档的定义过程中客户端可以从元素仓库中获取一系列的元素组装使用，例如定义用户任务(userTask)等。该阶段所有的元素均处于 XML 定义阶段。

(2) 元素解析层。

BpmnXMLConverter 类中的 convertToBpmnModel 方法所做的工作就是将流程文档中的元素解析并转化为 Activiti 中的内部表示 BaseElement 实例，并最终使用流程模型 BpmnModel 实例进行保存（类似内存数据库），从而使所有的元素解析结果均可以通过 BpmnModel 实例对象进行获取，而无需再次解析流程文档，提高运行效率。元素解析工作完成之后，流程文档中的所有元素已经与 activiti-bpmn-model-5.21.0.jar 包中的 BaseElement 实例对象一一对应起来。

(3) 对象解析层。

7.3 节中 bpmnParse.execute() 方法的处理过程，就是将 BaseElement 实例对象中的属性信息解析并转化为流程虚拟机中可识别的 ActivityImpl 实例对象或者 TransitionImpl 实例对象。

接下来使用 userTask 节点的定义、元素解析和对象解析三个阶段来举例说明，流程文

档中的 userTask 元素经过元素解析层之后转换为 UserTask 实例对象, UserTask 实例经过对象解析层之后蜕变为 ActivityImpl 实例对象, UserTask 实例对象中的所有信息作为 ActivityImpl 实例对象的属性值存在, 这样后续操作可以直接通过流程虚拟机获取 ActivityImpl 实例对象, 进而获取到任务节点的所有信息。

元素解析与对象解析两阶段的工作完毕之后, 所有的 BaseElement 实例对象已经与流程虚拟机中 ActivityImpl 实例对象对应起来。在对象解析阶段, ActivityImpl 实例对象需要根据元素实体的类型填充不同的属性值, 以方便流程虚拟机按照元素的类型进行区分处理, 该阶段涉及的类包括但不限于流程虚拟机类 ActivityImpl、任务节点实体类 (TaskDefinition) 和活动行为类。

看到这里可能会有这样的疑问: 直接用元素解析层岂不是更简单, 为何需要对对象解析层呢? 因为流程文档定义的一系列元素经过元素解析层之后转变为不同的 BaseElement 实例对象, 如果没有对象解析层, 那么 BaseElement 类承载的功能太多了, 后续虚拟机所有的操作均需要对 BaseElement 实例进行区别对待, 最重要的一点就是不能全局统一管理, 引入了对象解析层后, 所有的 BaseElement 实例最终统一使用 ActivityImpl 实例对象或者 TransitionImpl 实例对象表示, 并且 ActivityImpl 实例对象和 TransitionImpl 实例对象两者之间可以相互获取, 还可以按照元素的类型添加不同的行为类。分层的目的是为了让每层的职责更加单一, 看起来也更加清晰, 代码也更加容易维护。

10.2 虚拟机入口

本节详细分析流程文档元素解析之后的对象注入流程虚拟机的过程, 流程虚拟机中的属性注入工作在流程资源部署时已经开始, 7.3 节中提到 BpmnDeployer 类中的 deploy 方法调用了 bpmnParse.execute() 方法, 该方法的具体实现, 如代码清单 10-1 所示。

代码清单 10-1 BpmnParse.java

```
1 public BpmnParse execute() {
2     try {
3         //首先获取 ProcessEngineConfigurationImpl 对象
4         ProcessEngineConfigurationImpl processEngineConfiguration =
5             Context.getProcessEngineConfiguration();
6         //实例化 BpmnXMLConverter 对象
7         BpmnXMLConverter converter = new BpmnXMLConverter();
8         boolean enableSafeBpmnXml = false;
9         String encoding = null;
10        if (processEngineConfiguration != null) {
11            enableSafeBpmnXml = processEngineConfiguration.isEnableSafeBpmnXml();
12            encoding = processEngineConfiguration.getXmlEncoding();
13        }
14        //将流程文档转化为 Activiti 内部表示
15        if (encoding != null) {
16            bpmnModel = converter.convertToBpmnModel(streamSource, validateSchema, enableSafeBpmnXml,
17                encoding);
18        }
19    }
20 }
```



```

18 } else {
19     bpmnModel = converter.convertToBpmnModel(streamSource, validateSchema, enable-
SafeBpmnXml);
20 }
21 if (validateProcess) {
22     //默认为 true 即开启流程验证
23     ...//省略流程文档格式验证代码
24 }
25 createImports();
26 createItemDefinitions();
27 createMessages();
28 createOperations();
29 transformProcessDefinitions();
30 }
31 return this;
32 }

```

将 execute 方法的核心处理逻辑总结如下。

(1) 元素转换。

第 6~19 行通过调用 BpmnXMLConverter 类的 convertToBpmnModel 方法将流程文档中的元素转换为 BaseElement 实例对象,所有元素转换后对应的对象均保存在 bpmnModel 对象中,第 4.3 节详细讲解过。

(2) bpmnModel 验证。

第 21 行如果 validateProcess 参数值为 true,则需要开启 BpmnModel 实例对象的验证工作,该参数值默认为 true,验证工作委托 ProcessValidatorImpl 类的 validate 方法进行完成,第 3.4 节详细讲解过。

(3) 第 25~28 行初始化一系列 WebService 服务所需要的元素对象。WebService 的相关知识,不作为本书讲解的重点。

(4) transformProcessDefinitions 方法用于全局调度对象解析工作。

该方法非常重要,负责全局调度对象的解析工作,其内部主要通过 BpmnParseHandler 接口的具体实现类完成对象的解析工作。

扩展

上述方法的主要工作是用于元素的解析和对象的解析,但第 25~28 行却用来初始化 WebService 服务所需要的元素对象。很显然这个地方的设计有点不合理,第 25~28 行代码放置在 WebServiceActivityBehavior 类中会更加的合理,即使用的时候再去解析从而提升性能。

10.3 流程定义转换准备

transformProcessDefinitions 方法负责调度对象的解析工作,接下来详细分析该方法的实现逻辑,如代码清单 10-2 所示。

代码清单 10-2 BpmnParse.java

```

1  protected void transformProcessDefinitions() {
2      sequenceFlows = new HashMap<String, TransitionImpl>(); //初始化集合
3      for (Process process : bpmnModel.getProcesses()) { //遍历所有的 process 对象
4          if (process.isExecutable()) { //如果 process 是可以执行的
5              bpmnParserHandlers.parseElement(this, process); //开始解析对象
6          }
7      }
8      if (!processDefinitions.isEmpty()) {
9          processDI(); //解析元素坐标信息
10     }
11 }

```

该方法主要做了一些辅助性的工作,真正的核心工作委托给 bpmnParserHandlers.parseElement(this, process)方法实现,下面简单概括 transformProcessDefinitions 方法的实现逻辑,并从中解释它所提供的功能。

(1) 第3~7行循环遍历 process 对象,如果 process 对象的 isExecutable 属性值为 true,则调用 bpmnParserHandlers.parseElement(this, process)方法解析流程对象,这行代码非常重要,作为对象解析工作的切入点。

(2) 第9行委托 processDI 方法解析流程文档中的元素坐标。

10.3.1 初始化对象解析器集合

bpmnParserHandlers.parseElement(this, process)方法负责 process 对象的解析工作,接下来详细分析 bpmnParserHandlers 对象的初始化过程,该对象的初始化工作非常重要,因为如果该对象为空,则无法进行对象的解析工作。第7.1.1节中提到过 getDefaultDeployers 方法获取默认部署器的处理过程,如代码清单 10-3 所示。

代码清单 10-3 ProcessEngineConfigurationImpl.java

```

1  protected Collection<> getDefaultDeployers() {
2      ...//省略部署器的初始化过程
3      //实例化 Bpmn 解析处理器
4      List<BpmnParseHandler> parseHandlers = new ArrayList<BpmnParseHandler>();
5      //集合的添加顺序跟部署器集合添加的顺序类似,前置、内置、后置
6      if (getPreBpmnParseHandlers() != null) {
7          parseHandlers.addAll(getPreBpmnParseHandlers());
8      }
9      parseHandlers.addAll(getDefaultBpmnParseHandlers());
10     if (getPostBpmnParseHandlers() != null) {
11         parseHandlers.addAll(getPostBpmnParseHandlers());
12     }
13     //实例化 BpmnParseHandlers 类
14     BpmnParseHandlers bpmnParseHandlers = new BpmnParseHandlers();
15     bpmnParseHandlers.addHandlers(parseHandlers); //将解析器注入 bpmnParseHandlers
16     ...//省略设置属性代码
17 }

```

该方法的实现逻辑并不复杂,但是很多人不理解其作用以及意义所在,而且如果仅从表面上理解很难看懂其设计意图,因此需要从全局的角度思考 Activiti 是如何完成对象的解析工作。

(1) 第 4~12 行初始化对象解析器容器。

对象解析器容器 `parseHandlers` 的初始化过程可以分为初始化前置对象解析器、初始化内置对象解析器、初始化自定义内置对象解析器、初始化后置对象解析器。这样设计的好处是前置和后置对象解析器的定义完全交给客户端,如果客户端没有手动干预对象解析器的初始化工作,那么引擎就使用系统内置的对象解析器进行处理, `parseHandlers` 集合使用 List 数据结构存储对象解析器,如果客户端分别自定义了前置和后置对象解析器,同一个对象的解析器存在多个,那么 Activiti 使用何种策略定位对象解析器呢? 姑且留下一个悬念,稍后加以说明。

(2) 第 14~15 行实例化 `BpmnParseHandlers` 类,并为其填充 `parseHandlers` 属性值。

10.3.2 初始化内置对象解析器

接下来,分析 `getDefaultBpmnParseHandlers` 方法的处理逻辑,如代码清单 10-4 所示。

代码清单 10-4 ProcessEngineConfigurationImpl.java

```

1  protected List<BpmnParseHandler> getDefaultBpmnParseHandlers() {
2      List<BpmnParseHandler> bpmnParserHandlers = new ArrayList<BpmnParseHandler>();
3      bpmnParserHandlers.add(new BoundaryEventParseHandler());
4      ...//省略一系列对象解析器的添加过程
5      if (customDefaultBpmnParseHandlers != null){
6          Map<Class<?>, BpmnParseHandler> customParseHandlerMap = new HashMap();
7          for (BpmnParseHandler bpmnParseHandler : customDefaultBpmnParseHandlers) {
8              for (Class<?> handledType : bpmnParseHandler.getHandledTypes()){
9                  customParseHandlerMap.put(handledType, bpmnParseHandler);
10             }
11         }
12         for (int i = 0; i < bpmnParserHandlers.size(); i++) {
13             BpmnParseHandler defaultBpmnParseHandler = bpmnParserHandlers.get(i);
14             if (defaultBpmnParseHandler.getHandledTypes().size() != 1) {
15                 StringBuilder supportedTypes = new StringBuilder();
16                 for (Class<?> type : defaultBpmnParseHandler.getHandledTypes()) {
17                     supportedTypes.append(" ").append(type.getCanonicalName()).append(" ");
18                 }
19             } else {
20                 Class<?> handledType = defaultBpmnParseHandler.getHandledTypes().iterator().next();
21                 if (customParseHandlerMap.containsKey(handledType)) {
22                     BpmnParseHandler newBpmnParseHandler = customParseHandlerMap.get(handledType);
23                     bpmnParserHandlers.set(i, newBpmnParseHandler);
24                 }
25             }
26         }
27     }

```

```
28     for (BpmnParseHandler handler : getDefaultHistoryParseHandlers()){
29         bpmnParserHandlers.add(handler);
30     }
31     return bpmnParserHandlers;
32 }
```

根据上面的代码可知,对象解析器的初始化经历了如下几个过程。

(1) 第3~4行实例化内置对象解析器并将其添加到 bpmnParserHandlers 集合,这一系列的内置对象解析器均间接或者直接实现了 BpmnParseHandler 接口。

(2) 第5行判断 customDefaultBpmnParseHandlers 开关属性值是否为空,如果不为空则循环遍历该集合并替换步骤(1)中的内置对象解析器,该判断非常重要,如果开发人员觉得 Activiti 中的内置对象解析器不能满足自己的业务需求,更有甚者是一个 Bug,则可以通过该开关属性值替换内置对象解析器,由此也确实体会到了 Activiti 的良苦用心。

(3) 第12~26行循环遍历 bpmnParserHandlers 集合,如果 customParseHandlerMap 集合中存在值,则替换内置对象解析器。

(4) 第28~29行获取历史解析器,然后将其添加到 bpmnParserHandlers 集合中。

需要获取的历史解析器有 UserTaskHistoryParseHandler、StartEventHistoryParseHandler、ProcessHistoryParseHandler 等,添加历史解析器的目的主要是为对部分对象(这些对象均被历史解析器所管理)自动添加系统内置记录监听器,以上四种历史解析器被添加到 bpmnParserHandlers 集合中,这样解析 BaseElement 实例对象时,就会查找 BaseElement 实例对象对应的所有解析器,然后依次执行。通过分析上面 bpmnParserHandlers 集合元素的添加顺序,可以看出历史解析器永远是最后执行的(前提是开发人员没有定义后置对象解析器),可以查看这些历史解析器中 executeParse 方法的处理逻辑,该方法的处理逻辑相对比较简单,可以参考第11.3节。

使用图10-2对对象解析器的初始化过程进行通俗易懂的描述。

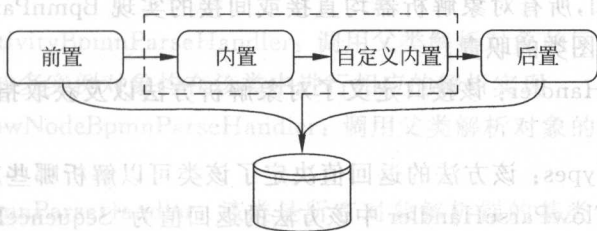


图 10-2 对象解析器初始化先后顺序

图10-2中前置对象解析器、自定义内置对象解析器、后置对象解析器三种类型的对象解析器,引擎均提供了开关方便客户端操作。以上所有的对象解析器最终使用 List 集合进行存储,这样同一个对象可以存在多个对象解析器对其进行解析处理,这一点有别于元素解析器的处理,元素解析器使用 Map 数据结构,一个元素只能存在一个元素解析器对其进行解析处理,所以对象解析器留给客户端的选择更多,但也侧面说明了操作对象解析器的复杂度远远大于元素解析器。

10.3.3 解析调度类 BpmnParseHandlers

从全局的角度了解了对象解析器集合的初始化工作之后,下面分析对象解析调度类 BpmnParseHandlers,该类的功能职责如图 10-3 所示。

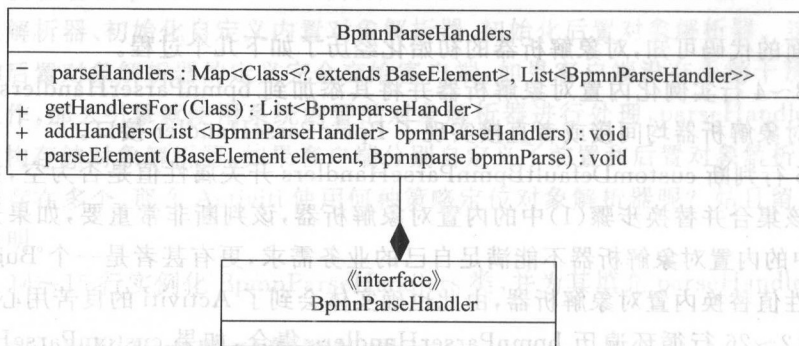


图 10-3 BpmnParseHandlers 类职责图

通过图 10-3 可以看出 BpmnParseHandlers 类负责对象解析器的全局统筹调用工作,其内部持有 parseHandlers 集合维护所有的对象解析器,该类提供了如下几个方法:

- (1) addHandlers: 该方法负责添加对象解析器。
- (2) parseElement: 该方法非常重要,负责根据对象的类型查找对象解析器并调度解析器完成对象解析工作。
- (3) getHandlersFor: 该方法负责查找对象解析器集合。

10.3.4 BpmnParseHandler 架构

在详细分析对象解析之前,先了解 BpmnParseHandler 类的结构,如图 10-4 所示。

根据图 10-4 可知,所有对象解析器均直接或间接的实现 BpmnParseHandler 接口,下面从全局角度讲解上图类的职责。

(1) BpmnParseHandler: 该接口定义了对象解析方法以及获取指定对象解析器的方法,如下所示。

- getHandledTypes: 该方法的返回值决定了该类可以解析哪些对象,例如连线解析器 SequenceFlowParseHandler 中该方法的返回值为“SequenceFlow.class”,这样需要解析 SequenceFlow 实例时,就可以直接使用 SequenceFlowParseHandler 类对其进行解析。
- parse(BpmnParse bpmnParse, BaseElement element): 解析 BaseElement 实例对象,根据 BaseElement 实例类型创建不同的 ActivityImpl 实例对象以及行为类实例对象,并最终将解析结果存放到 bpmnParse 对象中,这样后续可以直接从 bpmnParse 对象中获取已经解析过的对象,无须再次重复解析。bpmnParse 对象为 BpmnParse 类型,该对象存储了对象解析的公共属性值,因此可以将 BpmnParse 类理解为对象解析的上下文类。

(2) FlowNodeHistoryParseHandler: 负责历史节点对象的解析处理工作。该类主要

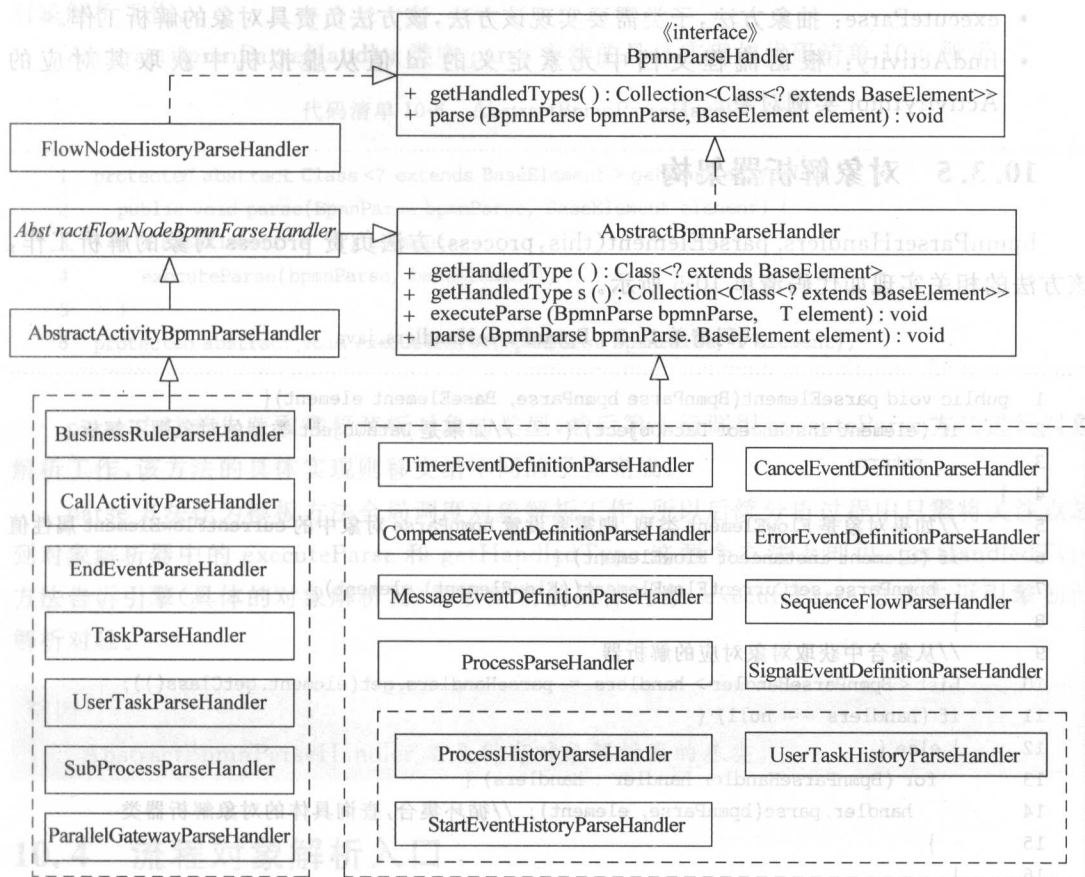


图 10-4 BpmnParseHandler 类的结构

负责为其内部持有的 supportedElementClasses 集合中的对象自动注入不同的系统内置记录监听器。

(3) AbstractActivityBpmnParseHandler: 调用父类解析对象的同时, 添加了多实例对象的解析工作, 所有的多实例对象均在该类中进行相应的解析实现。

(4) AbstractFlowNodeBpmnParseHandler: 调用父类解析对象的同时, 负责执行监听器的添加工作。

(5) AbstractBpmnParseHandler: 该类是所有对象解析器的基类, 定义了解析对象的模板方法, 对象的具体解析工作则移植到不同的子类进行实现, 该类还提供了如下几个重要的方法。

- createActivityOnScope: 创建 ActivityImpl 实例对象。
- createExecutionListener: 创建执行监听器实例对象。
- createExecutionListenersOnScope: 为解析的对象添加执行监听器支持。
- createIOSpecification: 创建 IOSpecification 实例对象。
- getHandledTypes: 获取对象集合。
- getHandledType: 该方法为抽象方法, 子类需要对其进行实现。
- parse: 该方法作为对象解析的模板方法存在, 全局调度对象解析工作。

- executeParse: 抽象方法, 子类需要实现该方法, 该方法负责具对象的解析工作。
- findActivity: 根据流程文档中元素定义的 id 值从虚拟机中获取其对应的 ActivityImpl 实例对象。

10.3.5 对象解析器架构

bpmnParserHandlers.parseElement(this, process)方法负责 process 对象的解析工作, 该方法的相关实现如代码清单 10-5 所示。

代码清单 10-5 BpmnParseHandlers.java

```

1 public void parseElement(BpmnParse bpmnParse, BaseElement element){
2     if (element instanceof DataObject) { //如果是 DataObject 类型的对象则不解析
3         return;
4     }
5     //如果对象是 FlowElement 类型,则需要设置 bpmnParse 对象中的 currentFlowElement 属性值
6     if (element instanceof FlowElement) {
7         bpmnParse.setCurrentFlowElement((FlowElement) element);
8     }
9     //从集合中获取对象对应的解析器
10    List<BpmnParseHandler> handlers = parseHandlers.get(element.getClass());
11    if (handlers == null) {
12    } else {
13        for (BpmnParseHandler handler : handlers) {
14            handler.parse(bpmnParse, element); //循环集合, 查询具体的对象解析器类
15        }
16    }
17 }

```

下面概括一下 parseElement 方法的处理逻辑, 并从中解释它所提供的功能。

(1) 解析对象之前的准备工作: 判断 element 对象的类型。

第 2 行如果对象为 DataObject 类型, 则第 3 行方法直接返回, 不会进行后续处理; 第 6 行如果对象为 FlowElement 类型, 则第 7 行需要设置 bpmnParse 对象的 currentFlowElement 属性值为当前准备解析的 element 对象, 这样设计的意图在于, 所有的对象解析器最终都会调用 parseElement 方法(模板方法), 在当前对象解析之前, 将其保存在 bpmnParse 对象中, 下一个对象解析时可以从 bpmnParse 对象中获取到上一个对象的解析结果, 例如现在对 a、b、c 三个对象按照顺序进行解析, 这样解析 a 对象之前将 a 设置到 currentFlowElement 属性中(这里的 a 对象为引用类型, 解析完毕属性就被填充了), b 对象解析时就可以获取 a 对象的解析结果, 同理 c 对象解析时就可以获取 b 对象的解析结果。

(2) 第 10 行获取对象对应的解析器。

因为所有的对象解析器都存储在 parseHandlers 集合中, 所以根据对象的类型, 可以很轻松的获取该对象的解析器集合。parseHandlers 集合为 Map 数据结构, 因此这里的获取操作是非常高效的。

(3) 委托解析器进行解析工作。

第 13~15 行循环遍历 handlers 集合, 然后依次调用 handler 对象中的 parse 方法进行

对象解析工作。

AbstractBpmnParseHandler 类中 parse 方法的具体实现如代码清单 10-6 所示。

代码清单 10-6 AbstractBpmnParseHandler.java

```
1 protected abstract Class<? extends BaseElement> getHandledType();
2 public void parse(BpmnParse bpmnParse, BaseElement element) {
3     T baseElement = (T) element;
4     executeParse(bpmnParse, baseElement);
5 }
6 protected abstract void executeParse(BpmnParse bpmnParse, T element);
```

parse 方法首先获取需要解析对象的类型,然后第 4 行调用 executeParse 方法进行对象解析工作,该方法的具体实现则移交给不同的子类完成。

parse 方法作为模板方法全局调度对象解析工作,所以后续分析过程中只需将关注点放到对象解析器中的 executeParse 和 getHandledType 这两个方法上即可,getHandledType 方法告诉引擎(具体的对象解析器)需要解析的具体对象,executeParse 方法告诉引擎如何解析对象。

强调

AbstractBpmnParseHandler 类是所有对象解析器的基类。

10.4 流程对象解析入口

因为流程文档中大部分元素是 process 元素的子元素,所以这里以 process 元素的属性承载类 Process 为入口,探究 Activiti 对象解析的内部实现机制,Process 实例对象的解析过程如代码清单 10-7 所示。

代码清单 10-7 ProcessParseHandler.java

```
1 protected void executeParse(BpmnParse bpmnParse, Process process) {
2     if (process.isExecutable() == false) {...//省略日志输出代码}
3 } else {
4     bpmnParse.getProcessDefinitions().add(transformProcess(bpmnParse, process));
5 }
6 }
```

上面代码仅仅是在 Process 实例对象真正解析工作开始之前做了一层判断,第 2 行如果 process 对象的 executable 属性值为 false,则不进行对象解析工作,因为对不会执行的 process 对象进行解析是没有意义的;如果 process 对象的 executable 属性值为 true,则第 4 行调用 transformProcess 方法解析 process 对象,解析工作完毕,将该方法的返回值添加到 bpmnParse 对象的 processDefinitions 集合中。transformProcess 方法的相关实现如代码清单 10-8 所示。

代码清单 10-8 ProcessParseHandler.java

```

1  protected ProcessDefinitionEntity transformProcess(BpmnParse bpmnParse, Process process) {
2      //流程定义实体属性承载类
3      ProcessDefinitionEntity currentProcessDefinition = new ProcessDefinitionEntity();
4      //设置到 bpmnParse 对象中
5      bpmnParse.setCurrentProcessDefinition(currentProcessDefinition);
6      currentProcessDefinition.setKey(process.getId());
7      currentProcessDefinition.setName(process.getName());
8      //targetNamespace 属性值作为流程定义的分类使用
9      currentProcessDefinition.setCategory(bpmnParse.getBpmnModel().getTargetNamespace());
10     currentProcessDefinition.setDescription(process.getDocumentation()); //文档信息
11     currentProcessDefinition.setProperty("documentation", process.getDocumentation());
12     //暂时还没有开始解析任务节点,先实例化 HashMap,防止该属性值为空,程序报错;
13     currentProcessDefinition.setTaskDefinitions(new HashMap<String, TaskDefinition>());
14     //设置部署 id
15     currentProcessDefinition.setDeploymentId(bpmnParse.getDeployment().getId());
16     //创建执行监听器
17     createExecutionListenersOnScope(bpmnParse, process.getExecutionListeners(),
18     currentProcessDefinition);
19     //创建事件监听器
20     createEventListeners(bpmnParse, process.getEventListeners(), currentProcessDefinition);
21     //获取表达式管理器
22     ExpressionManager expressionManager = bpmnParse.getExpressionManager();
23     //循环遍历 Process 中的启动人
24     for (String candidateUser : process.getCandidateStarterUsers()) {
25         currentProcessDefinition.addCandidateStarterUserIdExpression(expressionManager.
26         createExpression(candidateUser));
27     } //循环遍历 Process 中的启动候选人组
28     for (String candidateGroup : process.getCandidateStarterGroups()) {
29         currentProcessDefinition.addCandidateStarterGroupIdExpression(expressionManager.
30         createExpression(candidateGroup));
31     }
32     //设置到 bpmnParse 对象中
33     bpmnParse.setCurrentScope(currentProcessDefinition);
34     //开始解析该流程下的所有元素对象
35     bpmnParse.processFlowElements(process.getFlowElements());
36     //解析 createAssociation
37     processArtifacts(bpmnParse, process.getArtifacts(), currentProcessDefinition);
38     //解析 DataObjects 实例对象并作为流程实例的变量使用
39     Map<String, Object> variables = processDataObjects(bpmnParse, process.getDataObjects(),
40     currentProcessDefinition);
41     //获取变量信息
42     if (null != currentProcessDefinition.getVariables()) {
43         currentProcessDefinition.getVariables().putAll(variables);
44     } else {
45         //设置变量值
46         currentProcessDefinition.setVariables(variables);
47     }
48     bpmnParse.removeCurrentScope();
49     if (process.getIoSpecification() != null) {

```

```
50 //获取 IOSpecification 对象并设置到 currentProcessDefinition 对象中
51 IOSpecification ioSpecification = createIOSpecification(bpmnParse,
52 process.getIoSpecification());
53 currentProcessDefinition.setIoSpecification(ioSpecification);
54 }
55 return currentProcessDefinition;
56 }
```

虽然该方法的代码量很多,但其处理逻辑非常清晰,将其梳理总结如下。

(1) 实例化 ProcessDefinitionEntity 类并为其填充属性。

首先第 3 行实例化 ProcessDefinitionEntity 类,然后将流程文档元素转换之后的 process 对象赋值到 currentProcessDefinition 对象中,常用的属性有 id、name、documentation、部署 id、候选人或者组等。

(2) 创建执行监听器实例对象。

第 17~18 行 createExecutionListenersOnScope 方法用于创建执行监听器实例对象,可以参考第 11.4 节。

(3) 创建事件监听器实例对象。

第 20 行 createEventListeners 方法用于创建事件监听器实例对象,从这里也可以看出来只有 process 元素才可以配置全局事件监听器,其他元素不可以定义事件监听器。

(4) 第 22 行获取表达式对象 expressionManager。

(5) 第 24~31 行设置 currentProcessDefinition 对象中的 candidateStarterUserIdExpressions 和 candidateStarterGroupIdExpressions 属性值。

(6) 第 33 行将 currentProcessDefinition 对象设置到 bpmnParse 对象中。

(7) 解析 process 中所有的子元素对象。

第 35 行 bpmnParse.processFlowElements(process.getFlowElements()) 方法用于解析 process 对象中所有的子元素对象,也就是解析该流程下所有元素的属性承载类实例对象。流程中所有的子元素对象信息均可以通过 process.getFlowElements() 方法进行获取。

(8) 填充 bpmnParse 对象属性。

BpmnParse 类作为解析 process 对象的入口,所有的对象解析器均需要使用该类,该类承载 process 对象解析入口的同时也承载了公共属性的存取。上面代码中第 33 行操作设置了 bpmnParse 对象的 currentScopeStack 属性值,该操作非常重要(其内部使用 LinkedList 数据结构作为栈使用),这里必须要牢记一点,解析 process 对象时,首先将 currentProcessDefinition 对象设置到 bpmnParse 对象的 currentScopeStack 属性中,然后再解析 process 子元素对象,子元素对象解析时可以直接从 bpmnParse 对象的 currentScopeStack 属性中获取 ProcessDefinitionEntity 实例对象,所有的子元素对象解析完毕,执行第 48 行调用 bpmnParse.removeCurrentScope() 方法移除 currentScopeStack 集合中的元素,稍后详细讲解 BpmnParse 类的架构设计。

(9) 第 39~47 行对 dataObject 元素的属性承载类实例对象进行解析。

该元素的定义和使用可以参考第 13.3.2 节。

(10) 第 49~54 行流程变量与 WebService 的输入输出参数进行相互转换,输入输出参数的配置如代码清单 10-9 所示。

代码清单 10-9 webservice. bpmn

```

1 <dataInputAssociation>
2   <sourceRef> shareniuInput </sourceRef>
3   <targetRef> shareniuNum </targetRef>
4 </dataInputAssociation>
5 <dataOutputAssociation>
6   <sourceRef> shareniuNum </sourceRef>
7   <targetRef> shareniuOutput </targetRef>
8 </dataOutputAssociation>

```

10.5 流程子元素对象解析入口

上面代码中 bpmnParse. processFlowElements(process. getFlowElements()) 方法用于解析 process 元素下所有子元素对象(如开始节点、任务节点、结束节点等),该方法的相关实现如代码清单 10-10 所示。

代码清单 10-10 BpmnParse. java

```

1 public void processFlowElements(Collection<FlowElement> flowElements) {
2     //所有的连线信息集合
3     List<SequenceFlow> sequenceFlowToParse = new ArrayList<SequenceFlow>();
4     //所有的边界事件集合
5     List<BoundaryEvent> boundaryEventsToParse = new ArrayList<BoundaryEvent>();
6     //所有的元素对象信息
7     List<FlowElement> deferredFlowElementsToParse = new ArrayList<FlowElement>();
8     //遍历所有的元素对象
9     for (FlowElement flowElement : flowElements) {
10         if (flowElement instanceof SequenceFlow) { //如果对象为连线对象
11             sequenceFlowToParse.add((SequenceFlow) flowElement);
12         } else if (flowElement instanceof BoundaryEvent) { //如果对象为边界事件
13             boundaryEventsToParse.add((BoundaryEvent) flowElement);
14         } else if (flowElement instanceof Event) {
15             deferredFlowElementsToParse.add(flowElement); //如果对象为事件
16         } else {
17             bpmnParserHandlers.parseElement(this, flowElement); //以上三者都不是开始解析
18         }
19     }
20     //遍历 deferredFlowElementsToParse 集合并对集合中元素进行解析
21     for (FlowElement flowElement : deferredFlowElementsToParse) {
22         bpmnParserHandlers.parseElement(this, flowElement);
23     }
24     for (BoundaryEvent boundaryEvent : boundaryEventsToParse) {
25         bpmnParserHandlers.parseElement(this, boundaryEvent);
26     }

```

```

27  for (SequenceFlow sequenceFlow : sequenceFlowToParse) {
28      bpmnParserHandlers.parseElement(this, sequenceFlow);
29  }
30  }

```

根据上面代码可以看出 process 对象的解析与其子元素对象的解析处理逻辑差异还是很大的,将以上代码的处理逻辑进行如下总结。

(1) 第 3~7 行初始化解析对象集合。

(2) 第 9~19 行遍历 process 对象中所有的子元素对象,并按照对象的类型进行归类以及解析,其中第 11 行将 SequenceFlow 类型的对象存储到 sequenceFlowToParse 集合,第 13 行将 BoundaryEvent 类型的对象存储到 boundaryEventsToParse 集合,第 15 行将 Event 类型的对象存储到 deferredFlowElementsToParse 集合,除此之外的所有常规活动对象直接委托 parseElement 方法进行对象的解析工作(如 userTask 节点)。从这里可以看出子元素对象的解析有先后顺序区分,回想一下元素的类型,例如任务节点对象不会依赖其他对象,但是连线元素就需要依赖源节点和目标节点,可以结合流程文档中元素定义的先后顺序进行思考,例如下面的一个场景:程序需要解析连线元素对象,那么肯定需要获取连线对象的源和目的地两个节点的对象信息,如果没有区分流程对象解析的先后顺序势必会造成同一节点对象可能被反复解析,而且解析顺序混乱,如果对元素对象的类型进行区分归档,这样就可以首先解析没有依赖关系的对象,然后再解析有依赖关系的对象,并将解析结果存储到 BpmnParse 实例对象中,这样后续解析对象时,可以直接从 BpmnParse 实例对象中获取已经解析的对象值而无须再次重复解析。

(3) 解析集合对象。

第 21~29 行按照指定的顺序分别遍历以上三个集合,并调用 parseElement 方法解析集合中的对象,附带的好处就是解除元素对象之间的相互依赖问题,真可谓一箭双雕。

10.5.1 任务节点对象解析器

由于 Activiti 流程文档中的元素对象非常多,所以对应的对象解析器也很多,并且一个对象可能有多个解析器,两者为一对多的关系,因为对象解析逻辑大体相同,每个类都进行讲解可能会有事倍功半的效果,所以本节以 UserTask 的解析器 UserTaskParseHandler 为例详细讲解任务节点的解析流程,如代码清单 10-11 所示。

代码清单 10-11 UserTaskParseHandler.java

```

1  protected void executeParse(BpmnParse bpmnParse, UserTask userTask) {
2      //创建 ActivityImpl 实例对象
3      ActivityImpl activity = createActivityOnCurrentScope(bpmnParse, userTask, "userTask");
4      activity.setAsync(userTask.isAsynchronous()); //设置任务节点的 asynchronous 属性
5      activity.setExclusive(!userTask.isNotExclusive()); //设置 notExclusive 属性值
6      TaskDefinition taskDefinition = parseTaskDefinition(bpmnParse, userTask, userTask.getId(),
7      (ProcessDefinitionEntity) bpmnParse.getCurrentScope().getProcessDefinition()); //解析配置信息
8      activity.setProperty("taskDefinition", taskDefinition); //设置 taskDefinition 属性值
9      activity.setActivityBehavior(bpmnParse.getActivityBehaviorFactory().

```



```

10     createUserTaskActivityBehavior(userTask, taskDefinition)); //添加任务节点的行为类
11 }

```

UserTaskParseHandler 类中 executeParse 方法的处理逻辑非常清晰,将其梳理总结如下。

(1) 第 3 行创建 ActivityImpl 实例对象。

该类为所有元素对象(连线元素对象除外)在流程虚拟机中的内部表示。ActivityImpl 类是流程虚拟机中非常重要的一个类,主要存储节点的名称、类型、描述、监听器等信息,该类的实例对象由 createActivityOnCurrentScope 方法负责创建,为何不直接在该类中进行对象的创建工作呢?这里说明一下,因为大部分对象解析时均需要创建该实例对象,所以该实例对象创建工作在 UserTaskParseHandler 类的基类 AbstractBpmnParseHandler 中提供了默认实现,很好的体现了代码复用原则,唯一需要区别的地方就是对象的类型以及属性而已。

(2) 第 4~8 行填充 ActivityImpl 实例对象的属性。

对于 userTask 对象在流程虚拟机中的内部表示 ActivityImpl 实例来说,该对象需要添加 asynchronous、notExclusive 以及 taskDefinition 等属性。parseTaskDefinition 方法是为了获取对象中(在这里指任务节点)已经存在的属性信息并将其设置到 ActivityImpl 实例对象的 properties 属性中,该方法位于 UserTaskParseHandler 类中,因为该方法主要负责解析 userTask 节点属性,所以完全没有必要将 parseTaskDefinition 方法放在 AbstractBpmnParseHandler 类中进行实现。setProperty 方法内部使用 Map 集合存储 taskDefinition 信息值,而不需要在 ActivityImpl 类中单独定义一个个属性进行存储,由于 Activiti 中元素的类型非常多,因此使用 Map 集合存储差异化的属性是最灵活的方式。

(3) 第 9~10 行添加行为类。

为任务对象添加行为类 UserTaskActivityBehavior,该操作非常的重要,决定了当前任务节点完成之后可以到达的目的地、途径的连线信息等。更多的知识点可以参考第 14 章。

接下来详细分析 createActivityOnCurrentScope 方法创建 ActivityImpl 实例的过程。

10.5.2 创建 ActivityImpl 实例对象

createActivityOnCurrentScope 方法位于 AbstractBpmnParseHandler 类中,具体实现如代码清单 10-12 所示。

代码清单 10-12 AbstractBpmnParseHandler.java

```

1 public ActivityImpl createActivityOnCurrentScope(BpmnParse bpmnParse, FlowElement flowElement,
2 String xmlLocalName) {
3     return createActivityOnScope(bpmnParse, flowElement, xmlLocalName,
4 bpmnParse.getCurrentScope());
5 }
6 public ActivityImpl createActivityOnScope(BpmnParse bpmnParse, FlowElement flowElement, String
7 xmlLocalName, ScopeImpl scopeElement) {
8     //这里的 scopeElement 对应 ProcessDefinitionEntity 类
9     ActivityImpl activity = scopeElement.createActivity(flowElement.getId());

```

```

10 bpmnParse.setCurrentActivity(activity); //设置当前解析的元素
11 activity.setProperty("name", flowElement.getName()); //设置 name 值
12 activity.setProperty("documentation", flowElement.getDocumentation());
    //设置描述信息
13 if (flowElement instanceof Activity) { //判断元素的类型 UserTask 是 Activity 子类
14     Activity modelActivity = (Activity) flowElement;
15     activity.setProperty("default", modelActivity.getDefaultFlow());
    //设置任务节点默认的出线
16     if(modelActivity.isForCompensation()) { //设置 isForCompensation 属性值
17         activity.setProperty("isForCompensation", true);
18     }
19 } else if (flowElement instanceof Gateway) { //如果元素是网关类型,则需要设置默认值
20     activity.setProperty("default", ((Gateway) flowElement).getDefaultFlow());
21 }
22 activity.setProperty("type", xmlLocalName); //向属性集合添加元素的类型,也就是元素的名称
23 return activity;
24 }

```

第 1~2 行定义的方法直接委托第 6~7 行定义的方法进行工作,第 4 行 bpmnParse.getCurrentScope() 操作就是获取 bpmnParse 对象中的 currentScopeStack 集合,关于这一点前面的讲解也反复提到过,该集合存储的是 ProcessDefinitionEntity 实例对象。下面将第 6~7 行定义的方法的实现逻辑进行总结。

(1) 第 9 行创建 ActivityImpl 实例对象。

(2) 第 10 行将 ActivityImpl 实例对象填充到 bpmnParse 对象中。

(3) 第 11~22 为 ActivityImpl 实例对象填充属性。

常用的属性有 name、documentation、type。如果元素类型为 Activity(“活动”)则需要设置 default 以及 isForCompensation 属性值,如果元素类型为 Gateway(“网关”)则需要设置 default 属性值。

上面两个方法均位于 AbstractBpmnParseHandler 抽象类中,因此所有的子类均可以复用该代码。上述两个方法全局统筹 ActivityImpl 类的实例化工作并为实例对象填充属性。在上述代码中,第 9 行调用 ScopeImpl 类的 createActivity 函数创建 ActivityImpl 实例对象,该方法的具体实现如代码清单 10-13 所示。

代码清单 10-13 ScopeImpl.java

```

1 Map<String, ActivityImpl> namedActivities;
2 List<ActivityImpl> activities;
3 public ActivityImpl createActivity(String activityId) {
4     //id 可以理解为流程文档中对应的元素 id,processDefinition 参数对应 ProcessDefinitionEntity
5     ActivityImpl activity = new ActivityImpl(activityId, processDefinition);
6     if (activityId != null) {
7         if (processDefinition.findActivity(activityId) != null) {
8             throw new PvmException("duplicate activity id " + activityId + "");
9         }
10        namedActivities.put(activityId, activity);

```

```

11 }
12 //this 表示 ProcessDefinitionEntity 实例
13 activity.setParent(this);
14 activities.add(activity); //添加到 activities 集合
15 return activity;
16 }
17 public ActivityImpl findActivity(String activityId) {
18     ActivityImpl localActivity = namedActivities.get(activityId);
19     if (localActivity != null) {
20         return localActivity;
21     }
22     for (ActivityImpl activity: activities) {
23         ActivityImpl nestedActivity = activity.findActivity(activityId);
24         if (nestedActivity != null) {
25             return nestedActivity;
26         }
27     }
28     return null;
29 }

```

ActivityImpl 实例对象的创建逻辑非常清晰,将其梳理总结如下。

(1) 第 5 行实例化 ActivityImpl 类。

根据 activityId 和 processDefinition 两个参数值实例化 ActivityImpl 类。ActivityImpl 类的构造方法会调用其父类 ScopeImpl 的构造方法,而 ScopeImpl 构造方法继续调用其父类 ProcessElementImpl 的构造方法,因此以上所说的两个参数值最终会存储到 ProcessElementImpl 类中。

(2) 操作 activity 对象。

第 6 行如果 activityId 参数值不为空,则开始进行下一步操作。

(3) 第 7 行根据 activityId 值调用 processDefinition 对象的 findActivity 方法进行下一步的处理。该过程非常简单,首先尝试从 namedActivities 集合中查找,如果查询到则直接返回;否则开始对 activities 集合进行遍历并查找,如果以上所述的两个集合均不存在值,该方法直接返回 null。如果第 7 行查询到 activityId,则程序直接报错,该操作是为了确保已经存在于 namedActivities 集合或者 activities 集合中的节点不会被覆盖掉。

(4) 操作 namedActivities 集合。

第 10 行将 activity 对象添加到 namedActivities 集合中。

(5) 属性填充。

第 13 行设置 ActivityImpl 实例对象的 parent 属性值为当前执行对象,也就是 ProcessDefinitionEntity 实例对象。通过该操作之后,所有的 ActivityImpl 实例对象与 ProcessDefinitionEntity 实例进行双向关联,可以相互获取,例如在流程文档部署时,可以通过引擎提供的 API 获取 ProcessDefinitionEntity 实例对象,然后通过该实例对象获取 ActivityImpl 实例对象。

(6) 第 14 行添加缓存。

ActivityImpl 实例对象的创建工作是非常消耗系统资源的,该实例对象创建完成之后,

需要将已经创建的实例对象添加到 `namedActivities` 和 `activities` 集合中进行缓存,方便后续查找,从第 17 行定义的 `findActivity` 方法处理逻辑也可以看出缓存该实例对象的意义所在。

以上所有步骤操作完毕,就可以很轻松的通过 `ProcessDefinitionEntity` 实例对象获取 `namedActivities` 或者 `activities` 集合。换言之,任务节点对象已经被成功被注入到流程虚拟机中了。

10.5.3 多实例对象解析

上文以 `UserTaskParseHandler` 解析器为例详细分析了任务对象的解析过程,对于任务节点的使用来说,多实例任务的使用场景也是非常多的,上文讲解过多实例元素的解析工作在 `AbstractActivityBpmnParseHandler` 抽象类中提供了默认实现。接下来,分析 `Activiti` 是如何解析多实例对象的,如代码清单 10-14 所示。

代码清单 10-14 `AbstractActivityBpmnParseHandler.java`

```
1 public void parse(BpmnParse bpmnParse, BaseElement element) {
2     super.parse(bpmnParse, element); //调用父类进行解析工作
3     //如果元素是多实例,开始解析多实例
4     if (element instanceof Activity && ((Activity) element).getLoopCharacteristics() != null) {
5         createMultiInstanceLoopCharacteristics(bpmnParse, (Activity) element);
6     }
7 }
8 protected void createMultiInstanceLoopCharacteristics(BpmnParse bpmnParse, Activity model-
Activity) {
9     //获取流程文档中多实例的配置信息
10    MultiInstanceLoopCharacteristics loopCharacteristics = modelActivity.getLoopCharacteristics();
11    MultiInstanceActivityBehavior miActivityBehavior = null;
12    //获取多实例配置的节点,如果任务节点还没有解析则直接报错
13    ActivityImpl activity = bpmnParse.getCurrentScope().findActivity(modelActivity.getId());
14    //...省略判断代码
15    //判断是并行还是串行多实例,并添加不同的行为类
16    if (loopCharacteristics.isSequential()) {
17        miActivityBehavior =
18        bpmnParse.getActivityBehaviorFactory().createSequentialMultiInstanceBehavior(
19        activity, (AbstractBpmnActivityBehavior) activity.getActivityBehavior());
20    } else {
21        miActivityBehavior =
22        bpmnParse.getActivityBehaviorFactory().createParallelMultiInstanceBehavior(
23        activity, (AbstractBpmnActivityBehavior) activity.getActivityBehavior());
24    }
25    //...省略设置 ActivityImpl 的 isScope 属性值为 true,后续多实例任务节点运转的时候需要区
别对待
26 }
```

将上述代码中多实例任务对象解析以及注入虚拟机的处理流程,梳理总结如下。

(1) 第 2 行委托父类解析任务节点。

因为多实例任务节点本质还是任务节点,所以首先需要对任务对象进行解析,解析完毕

如果该任务是多实例任务则进行如下操作。

(2) 如果 userTask 节点是多实例任务,则第 5 行委托 createMultiInstanceLoopCharacteristics 方法解析多实例对象并填充属性,下面细化讲解 createMultiInstanceLoopCharacteristics 方法的处理逻辑。

- 首先第 10 行实例化 MultiInstanceLoopCharacteristics 类。
- 第 13 行根据 userTask 的 id 值从 bpmnParse 实例对象中的 currentScopeStack 集合中查找 ActivityImpl 实例对象。
- 第 16~23 行根据多实例对象配置的串行或者并行策略实例化不同的行为类。

通过上文分析任务节点以及多实例任务节点的解析逻辑,发现多实例任务节点和非多实例任务节点最终都用 ActivityImpl 类型的对象进行表示,唯一不同的地方就是对象的属性值以及行为类。

扩展

任务节点与多实例任务节点可以通过 multiInstance 属性或者行为类进行区分。

10.5.4 连线对象解析

接下来,以 SequenceFlow 的解析器 SequenceFlowParseHandler 为例详细讲解 SequenceFlow 实例对象的解析过程,如代码清单 10-15 所示。

代码清单 10-15 SequenceFlowParseHandler.java

```

1  protected void executeParse(BpmnParse bpmnParse, SequenceFlow sequenceFlow) {
2      ScopeImpl scope = bpmnParse.getCurrentScope();
3      ActivityImpl sourceActivity = scope.findActivity(sequenceFlow.getSourceRef());
4      ActivityImpl destinationActivity = scope.findActivity(sequenceFlow.getTargetRef());
5      Expression skipExpression;
6      if (StringUtils.isEmpty(sequenceFlow.getSkipExpression())) {
7          ExpressionManager expressionManager = bpmnParse.getExpressionManager();
8          skipExpression = expressionManager.createExpression(sequenceFlow.getSkipExpression());
9      } else {
10         skipExpression = null;
11     }
12     TransitionImpl transition = sourceActivity.createOutgoingTransition(sequenceFlow.getId(),
13         skipExpression);
14     bpmnParse.getSequenceFlows().put(sequenceFlow.getId(), transition);
15     transition.setProperty("name", sequenceFlow.getName());
16     transition.setProperty("documentation", sequenceFlow.getDocumentation());
17     transition.setDestination(destinationActivity);
18     if (StringUtils.isEmpty(sequenceFlow.getConditionExpression())) {
19         Condition expressionCondition = new
20             UelExpressionCondition(sequenceFlow.getConditionExpression());
21         transition.setProperty("conditionText", sequenceFlow.getConditionExpression());
22         transition.setProperty("condition", expressionCondition);
23     }
24     createExecutionListenersOnTransition(bpmnParse, sequenceFlow.getExecutionListeners(),

```

```

25     transition);
26 }

```

将该方法的处理流程总结如下。

(1) 第 3~4 行分别获取连线节点对应的 sourceActivity(源)和 destinationActivity(目标)。

(2) 第 5~11 行获取连线中配置的跳过表达式,并进行处理。

(3) 第 12~13 行实例化 TransitionImpl 类,该类为 sequenceFlow 对象在流程虚拟机中的最终表示,接下来分析该类初始化的整个过程,如代码清单 10-16 所示。

代码清单 10-16 ActivityImpl.java

```

1  protected List<TransitionImpl> outgoingTransitions = new ArrayList<TransitionImpl>();
2  protected Map<String, TransitionImpl> namedOutgoingTransitions = new HashMap<String,
TransitionImpl>();
3  public TransitionImpl createOutgoingTransition(String transitionId, Expression skipExpression) {
4      TransitionImpl transition = new TransitionImpl(transitionId, skipExpression,
5      processDefinition);
6      transition.setSource(this);
7      outgoingTransitions.add(transition);
8      if (transitionId != null) {
9          if (namedOutgoingTransitions.containsKey(transitionId)) {
10             throw new PvmException("activity '" + id + "' has duplicate transition '" +
transitionId + "'");
11         }
12         namedOutgoingTransitions.put(transitionId, transition);
13     }
14     return transition;
15 }

```

将 createOutgoingTransition 方法的处理逻辑梳理如下。

- 首先第 4~6 行实例化 TransitionImpl 类,并为其填充 source 属性值,在本案例中 this 为代码清单 10-15 中的 sourceActivity 对象。
- 第 7 行将 transition 对象添加到 outgoingTransitions 集合中。
- 第 9 行判断 transitionId 参数值是否已经存在 namedOutgoingTransitions 集合中,如果存在则直接报错,否则第 12 行将其添加到 namedOutgoingTransitions 集合中。

(4) 第 14~22 行填充 transition 对象属性,常用的属性有 name、documentation、conditionText、condition。其中第 17 行设置 transition 对象中的 destination(目标)属性值。

(5) 第 24~25 行 createExecutionListenersOnTransition 方法主要用于添加连线中配置的执行监听器。

10.6 PvmProcessElement 类架构

全面了解了任务节点对象、多实例节点对象以及连线对象的解析工作之后,接下来分析 PvmProcessElement 类的功能架构,如图 10-5 所示。

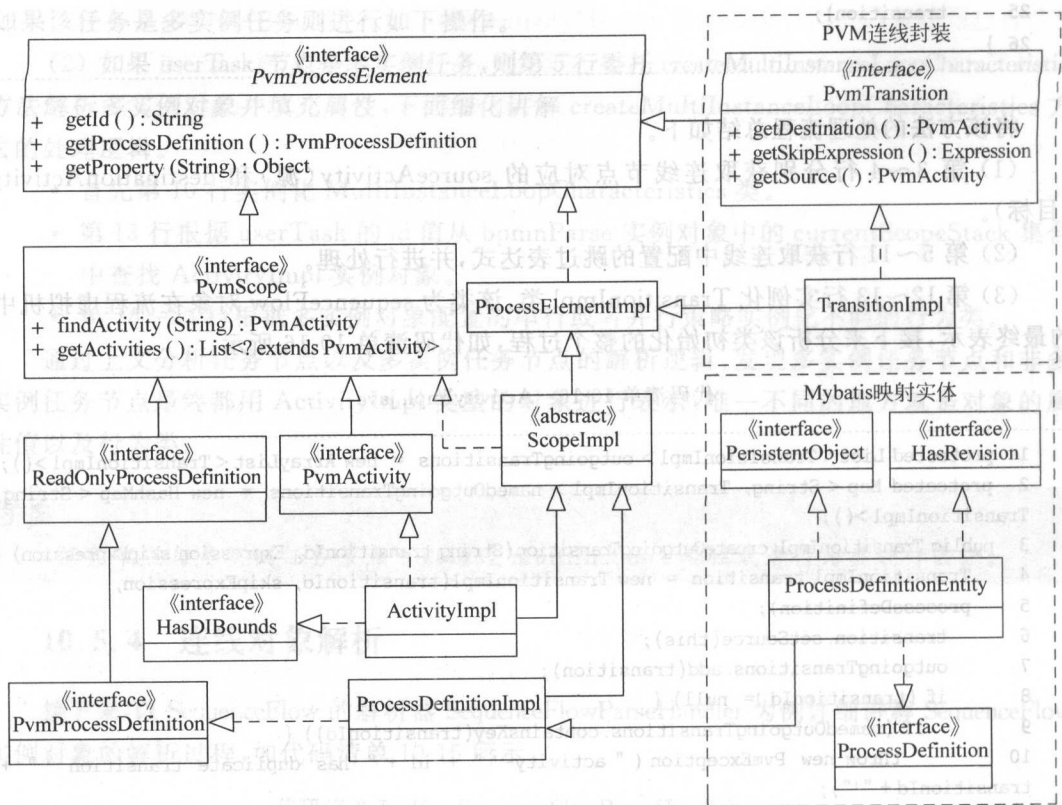


图 10-5 ScopeImpl 类的功能架构图

图 10-5 罗列了流程虚拟机中的大多数分类,下面大致讲解图中类的职责。

(1) `PvmProcessElement`: 定义了获取 `id` (流程文档中的元素 `id` 值)、获取 `PvmProcessDefinition` 实例、获取属性值的方法。关于属性设置可以参考上文的讲解。

(2) `ProcessElementImpl`: 对 `PvmProcessElement` 接口中的方法进行实现, 其内部使用 `HashMap` 数据结构维护 `properties` 集合, 该类在构造方法中初始化 `id` 和 `processDefinition` 两个属性值, 上文也提到过创建 `ActivityImpl` 对象的时候最终会调用该类的构造方法初始化 `id` 和 `processDefinition` 属性。

(3) PvmScope: 继承 PvmProcessElement 接口, 在 PvmProcessElement 接口定义功能的基础之上增加了获取所有 PvmActivity 实例对象的方法 getActivities, 以及根据 activityId 查找 PvmActivity 实例对象的方法 findActivity。

(4) PvmActivity: 继承 PvmScope, 在 PvmScope 定义功能的基础上增加了获取节点入线、出线的方法以及获取当前节点是否支持异步操作的方法。

(5) HasDIBounds: 提供了获取和设置元素的长度、宽度以及 X、Y 坐标信息的方法。元素的坐标以及长宽值,均存储在流程文档中,如果使用图形化工具设计流程文档,则绘制流程文档时会自动生成元素相应的坐标值以及长宽值。

(6) ScopeImpl: 对 PvmScope 接口中的部分方法进行实现。该类负责创建执行监听器等工作,其内部使用 Map 集合缓存 ActivityImpl 实例对象,更重要的是该类作为抽象类存在,子类均可以复用该类中的方法。

(7) ActivityImpl: 对接口 PvmActivity、HasDIBounds 中的方法进行实现,增加了创建节点对象的出线 and 入线方法以及获取对象的出线 and 入线方法。

(8) ReadOnlyProcessDefinition: 该接口继承 PvmScope,增加了对流程定义实体 key、name,描述信息的获取功能。

(9) PvmProcessDefinition: 继承 ReadOnlyProcessDefinition 接口,增加了获取流程部署 id 和创建流程实例的方法。

(10) ProcessDefinitionImpl: 对接口 PvmProcessDefinition 中定义的方法进行了实现。

(11) ProcessDefinitionEntity: 流程定义的映射实体,该类非常重要,间接继承了 ScopeImpl 类,因为所有对象解析之后的 ActivityImpl 实例对象均存储在 ScopeImpl 类中,所以可以通过 ProcessDefinitionEntity 实例对象获取所有对象解析之后的结果。

(12) HasRevision: 该接口定义了乐观锁的设置以及获取方法。

10.7 自定义对象解析器

有了前面的学习基础,现在要实现一个需求:对任务节点 userTask 对象进行扩展,在扩展元素中配置当前任务节点的处理人,这样流程运转到该任务节点时,就可以获取扩展元素的值,并将该值作为任务节点的处理人。为了简单起见直接使用任务监听器方式添加任务处理人,该需求在第 5 章中已经进行了实现,这里使用对象解析器的方式再次进行实现,仅仅是为了更加深入的了解对象解析器,在实现该需求之前,需要明白以下几个问题。

(1) 如何对任务节点 userTask 的扩展元素进行解析。

其实现比较简单,具体的做法就是自定义任务节点对象解析器,然后覆盖系统内置的解析器 UserTaskParseHandler。

(2) 如何存储扩展元素的值。

根据上文对 UserTaskParseHandler 类的讲解,可知 ActivityImpl 实例对象存储了活动节点的所有定义信息,所以可以在 userTask 元素解析时,通过 setProperty 方式将 userTask 元素的扩展值存储到 ActivityImpl 实例对象中。

(3) 如何获取扩展元素的值。

通过上文的学习可知 ActivityImpl 实例对象存储了所有节点的定义信息。换言之,只要能够获取到 ActivityImpl 实例对象,就可以通过该实例对象的 getProperty 方法获取所有的元素信息值。

下面根据上面的解题思路进行相关实现。

10.7.7 任务节点扩展属性

首先定义一个流程文档,该流程文档的内容如代码清单 10-17 所示,流程图如图 10-6 所示。

代码清单 10-17 pvm.bpmn20.xml

```
1 <process id="extensionOperationProcess" name="ext" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
```



```

3 <userTask id="operationTask" name="operationTask">
4   <extensionElements>
5     <activiti:taskListener event="create" class="com.shareniu.chapter10.TaskListener">
</activiti:taskListener>
6   <activiti:operations><activiti:transfer transferTo="shareniu" /></activiti:opera-
tions>
7   </extensionElements>
8 </userTask>
9 <sequenceFlow id="flow4" sourceRef="startevent1" targetRef="operationTask"></sequ-
enceFlow>
10 <userTask id="usertask2" name="usertask2"></userTask>
11 <sequenceFlow id="flow5" sourceRef="operationTask" targetRef="usertask2">
12 <conditionExpression xsi:type="tFormalExpression"><![CDATA[ ${shareniu} = 100 ]]]>
</conditionExpression>
13 </sequenceFlow>
14 <endEvent id="endevent1" name="End"></endEvent>
15 <sequenceFlow id="flow6" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>
16 </process>

```

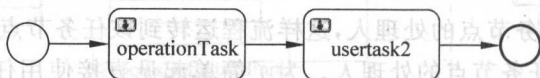


图 10-6 pvm.bpmn20.xml 对应的流程图

暂且将关注点放到 id 为 "operationTask" 的任务节点, 第 6 行定义了扩展元素, 其中 `<activiti:transfer>` 元素作为 `<activiti:operations>` 元素的子元素存在, `<activiti:transfer>` 元素中定义了 `transferTo` 属性, 该属性值为 "shareniu"。

第 5 行为 id 为 "operationTask" 的任务节点定义了一个任务监听器, 这样当流程运转到该节点时, 就可以通过 `ActivityImpl` 实例对象获取 `transferTo` 属性值, 并将该值作为当前任务节点的处理人。接下来定义一个类, 该类用于存储第 6 行定义的属性值, 具体实现如代码清单 10-18 所示。

代码清单 10-18 ExtensionOperation.java

```

1 public class ExtensionOperation
2   //存储自定义元素的属性和属性值
3   protected Map<String, String> propeies = new HashMap<String, String>();
4   /<activiti:transfer transferTo="shareniu" />中的 transfer 值
5   protected String name;

```

`<activiti:operations></activiti:operations>` 元素中, 可以定义很多扩展子元素, 虽然本案例只定义了一个子元素, 但是为了方便以后扩展子元素, 这里使用 `ExtensionOperation` 类承载子元素信息, 其中第 3 行定义的 `propeies` 为 `Map` 类型, 负责存储扩展元素的属性和属性值, 第 5 行定义的 `name` 为元素名称。

10.7.2 自定义任务节点对象解析器

为了简单起见, 自定义任务节点对象解析器 `ExtensionUserTaskParseHandler` 直接继


```

40 List<ExtensionAttribute> list2 = values2.next();
41 for ( ExtensionAttribute attributeElement : list2 ) {
42 //< activiti:transfer transferTo="shareniu" /> transferTo 作为 key, 值作为 value
43     userTaskOperation.addProperty( attributeElement.getName(),
44     attributeElement.getValue() );
45     //对应 transfer
46     operationMap.put( operationElement.getName(), userTaskOperation );
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 return(operationMap);
55 }
56 }

```

在上述代码中,第 8 行委托父类 `UserTaskParseHandler` 解析任务对象,关于这一点上文也讲解过,父类解析工作完毕,第 10 行就可以直接获取 `ActivityImpl` 实例对象,然后第 12 行调用 `parseInfo` 方法解析扩展元素,并将解析结果通过 `setProperty` 方法存储到 `ActivityImpl` 实例对象中。通过以上操作步骤,自定义扩展元素的值已经成功注入到 `ActivityImpl` 实例对象中。

10.7.3 获取自定义属性

`ExtensionUserTaskParseHandler` 类定义完毕,接下来的工作就是获取并操作自定义扩展属性,本案例使用任务监听器的方式,具体实现如代码清单 10-20 所示。

代码清单 10-20 TaskListener.java

```

1 public class TaskListener implements org.activiti.engine.delegate.TaskListener {
2     public void notify(DelegateTask delegateTask) {
3         ExecutionEntity ex = (ExecutionEntity) delegateTask.getExecution();
4         //根据执行对象获取所有的虚拟机对象
5         Map<String, Object> properties = ex.getActivity().getProperties();
6         //从集合中获取自定义的属性信息
7         Map<String, Object> map = (Map<String, Object>) properties.get("shareniu");
8         for (Object obj : map.keySet()) { //这里的获取与前面的存储过程相吻合
9             ExtensionOperation value = (ExtensionOperation) map.get(obj);
10            String val = value.getProperties().get("transferTo");//获取 transferTo 属性值
11            Collection<String> candidateUsers = new ArrayList<>();
12            candidateUsers.add(val);
13            delegateTask.addCandidateUsers(candidateUsers); //属性值作为当前任务的处理人
14        }
15    }
16 }

```

在上述代码中,第 3 行通过 `DelegateTask` 实例对象获取 `ExecutionEntity` 实例对象,然后第 5 行通过 `ExecutionEntity` 实例对象获取 `ActivityImpl` 实例对象,并通过 `ActivityImpl`

实例对象获取到自定义元素值。

第 13 行通过 `delegateTask.addCandidateUsers()` 方法将扩展的属性值添加到当前任务节点的处理人集合中。

上面代码的实现逻辑比较烦琐,而且极不灵活,如果流程文档中定义的任务节点比较少,这种实现方式勉强可以,如果流程文档中定义了大量的任务节点,难道要为每一个任务节点都配置一个任务监听器吗?显然很不合理,后续第 11.9 节会讲解一种更“高级”的用法,本案例的实现方法仅作为了解。

10.7.4 运用自定义对象解析器

接下来需要将 `ExtensionUserTaskParseHandler` 类注入流程引擎配置类,如代码清单 10-21 所示。当然也可以将自定义任务对象解析器作为后置对象解析器进行使用,该方式本节不做过多讲解,如果有兴趣可以自行实现。

代码清单 10-21 activiti.cfg.xml

```
1 <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
2   <property name="customDefaultBpmnParseHandlers">
3     <list>
4       <bean class="com.shareniu.chapter10.ExtensionUserTaskParseHandler" />
5     </list>
6   </property>
7 </bean>
```

在上述代码中,第 2~6 行将自定义对象解析器通过 `customDefaultBpmnParseHandlers` 开关属性注入流程引擎配置类。下面做一个简单的测试验证:首先部署 `pvm.bpmn20.xml` 流程文档,然后启动流程实例,因为流程实例启动之后,流程会直接运行到 `id` 为 `"operationTask"` 的任务节点,从而触发该任务节点配置的任务监听器,相关实现如代码清单 10-22 所示。

代码清单 10-22 ExtensionOperationProcessTest.java

```
1 public void addInputStreamTest() {
2   // 定义流程文档数据流
3   InputStream inputStream = DeploymentBuilderTest.class.getClassLoader().getResourceAsStream(
4     "com/shareniu/chapter10/pvm.bpmn20.xml");
5   // 构造 DeploymentBuilder 对象
6   DeploymentBuilder deploymentBuilder = repositoryService
7     .createDeployment().addInputStream(resourceName, inputStream);
8   // 部署
9   Deployment deploy = deploymentBuilder.deploy();
10 }
11 public void startProcessInstanceById() {
12   String processDefinitionId = "extensionOperationProcess:6:37504";
13   runtimeService.startProcessInstanceById(processDefinitionId);
14 }
```


在上述代码中,第 1 行定义的方法主要用于完成流程文档的部署工作,部署该流程文档的目的是为了重新启动新的流程实例并验证自定义对象解析器是否生效,该方法执行完毕,开始执行第 11 行定义的 `startProcessInstanceById` 方法,该方法主要用于启动流程实例,第 12 行 `processDefinitionId` 参数值为 `addInputStreamTest` 方法执行之后对应的流程定义 id 值,该值可以直接通过 `ACT_RE_PROCDEF` 表中的 `ID_` 列进行查找,执行完 `startProcessInstanceById` 方法后,流程实例会直接运转到 id 为 "operationTask" 的任务节点,并触发配置的任务监听器,以上操作执行完毕, `ACT_RU_IDENTITYLINK` 表中的新增数据如图 10-7 所示。

ID_	REV_	GROUP_ID_	TYPE_	USER_ID_
70005	1 (Null)		candidate	shareniu
70006	1 (Null)		participant	shareniu

图 10-7 ACT_RU_IDENTITYLINK 表数据的变化

扩展

自定义对象解析器也可以通过操作 `postBpmnParseHandlers` 开关属性注入流程引擎配置类。

10.8 流程虚拟机实战

10.8.1 获取流程虚拟机对象

上文反复提到 `ScopeImpl` 实例对象内部持有所有的 `ActivityImpl` 实例对象集合,下面讲解如何通过 `ProcessDefinitionEntity` 实例对象获取流程虚拟机中的对象,如代码清单 10-23 所示。

代码清单 10-23 ExtensionOperationProcessTest.java

```

1 public void findActivities() {
2     String processDefinitionId = "extensionOperationProcess:6:37504";
3     ProcessDefinitionEntity pdf = (ProcessDefinitionEntity)
4     repositoryService.getProcessDefinition(processDefinitionId);
5     List<ActivityImpl> activities = pdf.getActivities();
6     for (ActivityImpl activityImpl : activities) {
7         Object object = activityImpl.getProperties().get("shareniu");
8         if (object instanceof Map) {
9             Map<String, Object> map = (Map<String, Object>) object;
10            for (Object obj : map.keySet()) {
11                ExtensionOperation value = (ExtensionOperation) map.get(obj);
12            }
13        }
14        if (activityImpl.getId().equals("operationTask")) {
15            //获取 operationTask 节点的入线
16            List<PvmTransition> incomingTransitions = activityImpl.getIncomingTransitions();
17            System.out.println(incomingTransitions.get(0).getId());
18            //获取 operationTask 节点的出线
19            List<PvmTransition> outgoingTransitions = activityImpl.getOutgoingTransitions();
20            for (PvmTransition pvmTransition : outgoingTransitions) { //获取连线信息
21            }
22        }
23    }
24 }

```

```

22 }
23 }
24 }

```

在上述代码中,第2行 processDefinitionId 值为流程文档部署之后对应的流程定义 id 值,可以通过 ACT_RE_PROCDEF 表中的 ID_列进行查找,第3~4行根据 processDefinitionId 值获取 ProcessDefinitionEntity 实例对象,ProcessDefinitionEntity 类是 ProcessDefinition 接口的唯一实现类,因此强制类型转换不会报错,第5行根据 ProcessDefinitionEntity 实例对象的 getActivities 方法获取到所有的 ActivityImpl 实例对象,ActivityImpl 实例对象存储所有活动节点的信息供流程虚拟机运转时使用,第7~13行获取自定义元素的属性值,第14~22行获取任务节点的出线 and 入线等信息。

扩展

上述案例可以使用递归方式进行操作。

10.8.2 入侵流程虚拟机

上面讲解了如何获取流程虚拟机中的对象,接下来讲解如何修改流程虚拟机中的对象属性值,例如流程实例启动之前,获取 id 值为 operationTask 的任务节点,并将其对应的 ActivityImpl 实例对象中的任务节点名称进行修改,并在流程实例启动后观察数据库中的数据变化,该案例的相关实现如代码清单 10-24 所示。

代码清单 10-24 ExtensionOperationProcessTest.java

```

1 public void pvml() {
2     String processDefinitionId = "extensionOperationProcess:6:92503";
3     ProcessEngineConfigurationImpl pec = (ProcessEngineConfigurationImpl)
4     processEngine.getProcessEngineConfiguration();
5     DeploymentManager deploymentManager = pec.getDeploymentManager();
6     DeploymentCache<ProcessDefinitionEntity> processDefinitionCache =
7     deploymentManager.getProcessDefinitionCache();
8     ProcessDefinitionEntity pde = processDefinitionCache.get(processDefinitionId);
9     if (pde == null) {
10         pde = (ProcessDefinitionEntity) ((RepositoryServiceImpl) repositoryService)
11         .getDeployedProcessDefinition(processDefinitionId);
12     }
13     // 节点的 id 值
14     String taskDefKey = "operationTask";
15     ActivityImpl currActiviti = pde.findActivity(taskDefKey); // 当前活动节点
16     // 获取 TaskDefinition 实例对象
17     TaskDefinition task = (TaskDefinition) currActiviti.getProperty("taskDefinition");
18     ProcessEngineConfigurationImpl conf = (ProcessEngineConfigurationImpl) processEngine
19     .getProcessEngineConfiguration();
20     // 创建表达式
21     Expression expression = conf.getExpressionManager().createExpression("shareniu");
22     task.setNameExpression(expression); // 将表达式设置到任务节点中

```

```

23 processDefinitionCache.add(processDefinitionId, pde);
24 runtimeService.startProcessInstanceById(processDefinitionId);
25 }

```

将上面代码的处理逻辑进行如下总结。

- (1) 第 5 行获取 DeploymentManager 实例对象。
- (2) 第 8 行从缓存中获取 ProcessDefinitionEntity 实例对象，如果缓存数据丢失则 10~11 再次获取(该操作会重新生成缓存数据)。
- (3) 第 15 行获取 currActiviti 对象(id 为 operationTask 的任务节点)。
- (4) 第 17 行通过 currActiviti 对象获取 task 对象。
- (5) 第 21~22 行通过 conf 对象创建表达式 expression，然后将 expression 对象填充到 task 对象中。
- (6) 第 23 行将修改的数据更新到缓存。
- (7) 第 24 行启动流程实例。

执行上面的代码如不出意外，数据库 ACT_RU_TASK 表中新增数据如图 10-8 所示。

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	NAME	PROC DEF ID
95004	1	95001	95001	shareniu	extensionOperationProcess:6:92503

图 10-8 ACT_RU_TASK 表数据的变化

该案例使用了修改流程定义缓存的方式更新缓存中的数据，在实际项目开发中，同一个流程文档部署后对应的缓存数据是一个，而流程实例是多个，二者为一对多的关系，因此修改缓存数据的时候要特别谨慎，杜绝修改缓存中节点的出线或者入线信息，以防止误操作造成流程实例不可运行。

10.9 操作连线表达式

10.9.1 自动计算连线表达式

在实际项目开发中，期望获取连线元素配置的条件表达式，该如何实现？下面先分析 Activiti 是如何使用条件表达式的，如代码清单 10-25 所示。

代码清单 10-25 ExtensionOperationProcessTest.java

```

1 public void SimpleContext1() {
2     ExpressionFactory factory = new ExpressionFactoryImpl();
3     SimpleContext context = new SimpleContext();
4     context.setVariable("shareniu", factory.createValueExpression(10000, String.class));
5     ValueExpression e = factory.createValueExpression(context, "${shareniu >= 100}",
        boolean.class);
6     System.out.println(e.getValue(context));
7 }

```

在上述代码中,第2行实例化 ExpressionFactoryImpl 类,第3行实例化 SimpleContext 类,第4行为 context 设置变量,变量名称为"shareniu",值为10000,第5行设置需要计算的表达式,执行完以上代码,如不出意外,控制台的输出为 true。

10.9.2 获取连线表达式

代码清单 10-17 中,flow5 连线的源节点为 operationTask,因此如果期望获取 flow5 连线元素在虚拟机中的表示对象,那么则需要先获取 operationTask 在虚拟机中的表示对象,进而获取该对象的所有出线信息,具体实现如代码清单 10-26 所示。

代码清单 10-26 ExtensionOperationProcessTest.java

```
1 public void SimpleContext() {
2     String processDefinitionId = "extensionOperationProcess:6:37504";
3     ProcessDefinitionEntity pdf = (ProcessDefinitionEntity)
4     repositoryService.getProcessDefinition(processDefinitionId);
5     List<ActivityImpl> activities = pdf.getActivities();
6     for (ActivityImpl activityImpl : activities) {
7         if (activityImpl.getId().equals("operationTask")) {
8             List<PvmTransition> outgoingTransitions = activityImpl.getOutgoingTransitions();
9             PvmTransition destination = outgoingTransitions.get(0);
10            Object conditionText = destination.getProperty("conditionText");
11            Condition expressionCondition = (Condition) destination.getProperty("condition");
12            System.out.println(conditionText + ", " + expressionCondition);
13            ExpressionFactory factory = new ExpressionFactoryImpl();
14            SimpleContext context = new SimpleContext();
15            context.setVariable("shareniu", factory.createValueExpression(10000, String.class));
16            ValueExpression e = factory.createValueExpression(context, (String) conditionText,
17                boolean.class);
18            System.out.println(e.getValue(context));
19        }
20    }
21 }
```

在上述代码中,第10行获取连线元素配置的表达式字符串,第11行获取连线元素的表达式对象,第16行开始计算连线的条件是否成立。

10~11 再次获取(该操作不考) **听器音质理** 息时能出音两的录技新录而进

血力器训练

节点的任务处理人。

(2) 调用第三方系统,例如可以在监听器中发送邮件或者调用第三方的业务系统。调

(3) 历史节点信息入库,例如启动一个流程实例,则需要将开始节点的信息插入历史表

(4) 获取 Spring 中定义的 bean, 因为监听器运行过程中可能需要依赖项目中其他的服

上面一系列的场景,该如何使用监听器进行实现呢?监听器的类型以及支持的事件种

本章所述的监听器主要指的是执行监听器和任务监听器。

本章所述的监听器主要指的是执行监听器和任务监听器。

11.1 监听器生命周期

监听器从使用范围上可以划分为执行监听器和任务监听器,从功能实现上又可以细分为用户自定义监听器以及系统内置记录监听器(主要操作历史表,如记录活动节点的开始以及结束操作)。任务监听器仅支持作用于任务节点而执行监听器可以作用于流程三大要素等,两者之间仅仅是应用的节点范围以及类型不同而已,通常执行监听器的应用范围更加的广泛。任务监听器支持的事件类型有如下四种:节点分配处理人("assignment")、创建节点("create")、任务完成("complete")、任务删除("delete")。执行监听器支持的事件类型有如下三种:开始("start")、结束("end")、途径连线("take")。其中,"take"事件类型的执行监听器仅支持在连线中进行配置和使用,例如当前节点完成之后途径连线到达目标节点,那么就会触发连线上配置的执行监听器。需要注意一点,监听器的生命周期与之作用的节点或者连线的生命周期息息相关,紧密绑定,例如为任务节点 userTask1 配置了一个任务监听器,当流程实例运转到 userTask1 节点时,该任务节点的运行生命周期开始并触发状态变更,包括给任务节点分配处理人、创建任务节点、完成任务以及删除任务,状态变更则会触发相关事件类型的任务监听器,当任务节点执行完之后,流程实例继续向下运转,该任务节点的监听器运行生命周期伴随着作用的任务节点的结束而结束(除非当前流程实例再次流经该任务节点,如退回操作)。这里以第 9.2.3 节定义的流程文档为例,详细学习该流程中所有监听器执行的先后顺序如图 11-1 所示。

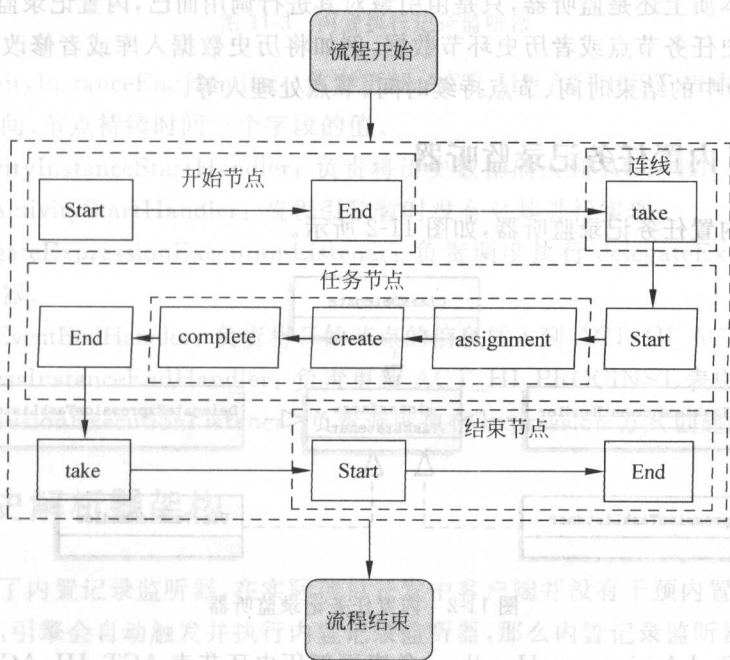


图 11-1 监听器执行的先后顺序

根据图 11-1 可以看出,流程实例永远都是以执行监听器为主线运行,如果流程实例运转的过程中发现了任务节点,则开始执行任务监听器,如果流程实例离开任务节点或者连

线,则该节点或者连线相对应的监听器的生命周期也随之结束(除非该任务节点或者连线再次被唤醒,如回退处理),任务监听器执行完毕之后再次触发执行监听器的 end 事件,推动流程实例继续向下运转。上图中也可以看出事件类型为 take 的执行监听器只可以在连线上进行配置,任务节点首先执行事件类型为 start 的执行监听器,然后开始执行所有的任务监听器(分配处理人、创建任务节点、完成任务节点),最后执行事件类型为 end 的执行监听器,换言之任务监听器的诞生只是为了方便对任务节点的操作而设计的,因为流程文档中任务节点的使用非常频繁。

任务监听器与执行监听器仅仅是实现行为接口类不同而已,任务监听器需要实现 TaskListener 接口,执行监听器需要实现 ExecutionListener 接口。

在 Activiti 中,对于监听器的创建可以使用如下三种方式。

(1) class: 指定类路径,指定的类需要实现 ExecutionListener 接口(执行监听器)或者 TaskListener 接口(任务监听器)。

(2) expression: 表达式方式创建,该方式在表达式中已经明确指定需要调用的类以及方法,并可以为需要调用的方法传入参数,形如 `${bean.doSomething(execution)}`。

(3) delegateExpression: 该方式通常与 Spring 框架配合起来使用。

11.2 内置记录监听器

Activiti 将运行数据与历史数据完全分开存储,也就是平时所说的运行表和历史表。内置记录监听器本质上还是监听器,只是由引擎对其进行调用而已,内置记录监听器的主要职责就是操作历史任务节点或者历史环节数据,例如将历史数据入库或者修改历史表(ACT_HI_ACTINST)中的结束时间、节点持续时间、节点处理人等。

11.2.1 内置任务记录监听器

首先讲解内置任务记录监听器,如图 11-2 所示。

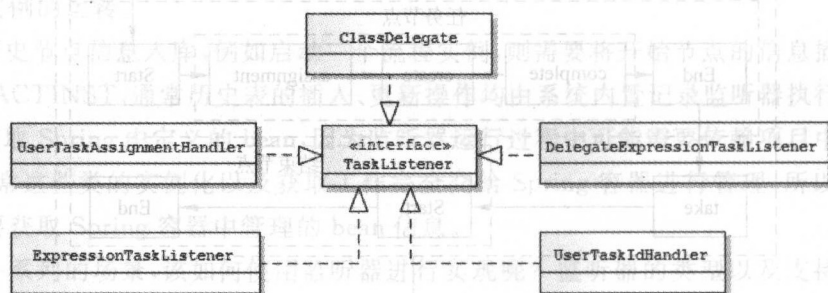


图 11-2 内置任务记录监听器

(1) UserTaskAssignmentHandler: 负责更新历史环节表 ACT_HI_ACTINST 中的任务节点处理人,对应该表的 ASSIGNEE_列。

(2) UserTaskIdHandler: 负责更新 ACT_HI_ACTINST 表中的任务 id 值。

(3) ClassDelegate: 对接口 TaskListener 和 ExecutionListener 中的方法进行实现,负

责调度执行 class 方式创建的监听器,其内部持有需要调用的执行监听器实例、任务监听器实例,并提供了实例化监听器的方法以及对实例对象进行属性填充的方法。

(4) DelegateExpressionTaskListener: 负责调度执行 delegateExpression 方式创建的任务监听器。

(5) ExpressionTaskListener: 负责调度执行 expression 方式创建的任务监听器。

11.2.2 内置执行记录监听器

内置执行记录监听器如图 11-3 所示。

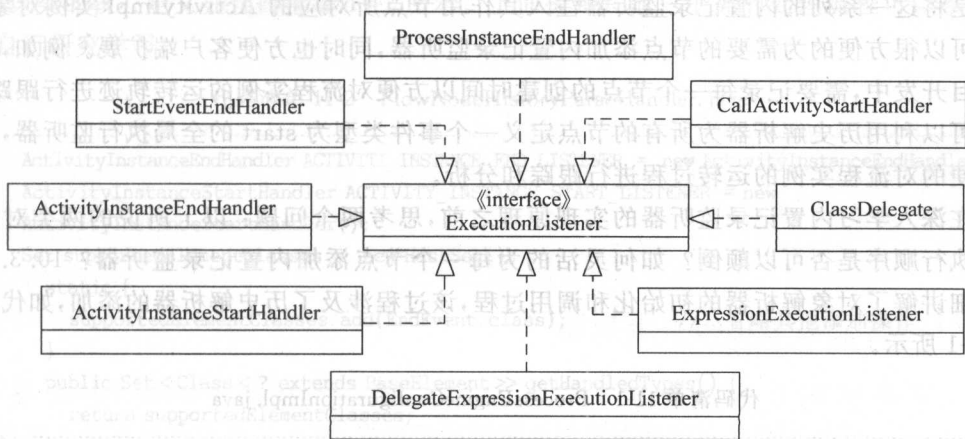


图 11-3 内置执行记录监听器

(1) ActivityInstanceEndHandler: 负责更新 ACT_HI_ACTINST 表中的删除原因(非必须)、结束时间、节点持续时间三个字段的值。

(2) ActivityInstanceStartHandler: 负责将历史数据插入到 ACT_HI_ACTINST 表。

(3) CallActivityStartHandler: 流程引擎暂时没有对其进行实现。

(4) DelegateExpressionExecutionListener: 负责调度执行 delegateExpression 方式创建的执行监听器。

(5) StartEventEndHandler: 负责将开始节点的信息插入到 ACT_HI_ACTINST 表中。

(6) ProcessInstanceEndHandler: 负责更新 ACT_HI_PROCINST 表中的数据。

(7) ExpressionExecutionListener: 负责调度执行 expression 方式创建的执行监听器。

11.3 历史解析器架构

上文讲解了内置记录监听器,在实际项目开发中客户端并没有干预内置记录监听器,流程实例运转时,引擎会自动触发并执行内置记录监听器,那么内置记录监听器是如何被添加到其作用的节点呢?内置记录监听器又完成了什么功能的呢?通过第 10 章的讲解可知可以存在多个对象解析器对 BaseElement 实例对象进行解析处理。通常流程三大要素在流程实例运转的过程中都会进行历史数据的归档操作,也就是操作平常所说的历史表,本书把负责历史归档的对象解析器统称为历史解析器,例如 userTask 任务节点需要进行历史数据的

归档,那么该任务节点势必存在多个对象解析器,第一类对象解析器负责将任务节点解析之后的信息注入流程虚拟机,第二类对象解析器也就是历史解析器,负责获取任务节点解析之后的结果 ActivityImpl 实例对象并为该对象自动注入内置记录监听器,只有这样设计,流程实例运转到该任务节点时才会自动触发不同事件类型的内置记录监听器。

为何要设计历史解析器?首先思考一个问题:内置记录监听器的作用,该类型的监听器完全是流程引擎为记录或者修改流程历史数据而设计的,所以没有必要让客户端干预和配置,以降低客户端使用的复杂度,附带的好处就是历史数据的归档操作完全交给内置记录监听器来完成,从而将流程实例运行数据与历史数据实现细节完全隔离。历史解析器的职责就是将这一系列的内置记录监听器注入其作用节点所对应的 ActivityImpl 实例对象,这样就可以很方便的为需要的节点添加内置记录监听器,同时也方便客户端扩展。例如,在实际项目开发中,需要记录每一个节点的创建时间以方便对流程实例的运转轨迹进行跟踪,这时就可以利用历史解析器为所有的节点定义一个事件类型为 start 的全局执行监听器,从而很方便的对流程实例的运转过程进行跟踪和分析。

在深入学习内置记录监听器的实现原理之前,思考两个问题:以上所说的两类对象解析器执行顺序是否可以颠倒?如何灵活的为每一个节点添加内置记录监听器?10.3.2 节中详细讲解了对对象解析器的初始化和调用过程,该过程涉及了历史解析器的添加,如代码清单 11-1 所示。

代码清单 11-1 ProcessEngineConfigurationImpl.java

```
1 protected List<BpmnParseHandler> getDefaultHistoryParseHandlers() {  
2     List<BpmnParseHandler> parseHandlers = new ArrayList<BpmnParseHandler>();  
3     parseHandlers.add(new FlowNodeHistoryParseHandler());  
4     parseHandlers.add(new ProcessHistoryParseHandler()); //负责事件、活动、网关内置记录监听器  
5     parseHandlers.add(new StartEventHistoryParseHandler()); //负责 process 元素记录监听器  
6     parseHandlers.add(new UserTaskHistoryParseHandler()); //负责 StartEvent 元素记录监听器  
7     return parseHandlers; //负责 UserTask 任务节点记录监听器  
8 }
```

在上述代码中,第 3~6 行实例化不同的历史解析器并将其添加到 parseHandlers 集合中。这些历史解析器负责为指定的 BaseElement 实例对象添加指定事件类型的内置记录监听器。

扩展

历史解析器本质上依然是对象解析器。

11.3.1 添加内置记录监听器

接下来,分析以上所说的四个历史解析器的职责。

(1) ProcessHistoryParseHandler: 负责解析 Process 实例对象并为其添加事件类型为

end 的内置记录监听器。

(2) StartEventHistoryParseHandler: 负责解析 StartEvent 实例对象并为其添加事件类型为 end 的内置记录监听器。

(3) UserTaskHistoryParseHandler: 负责解析 UserTask 实例对象并为其添加事件类型为 assignment 和 create 的内置记录监听器。

(4) FlowNodeHistoryParseHandler: 负责解析流程三大要素并为其添加事件类型为 start 和 end 的内置记录监听器。

下面以 FlowNodeHistoryParseHandler 类为例,详细说明整个内置记录监听器的添加过程,该类的核心定义如代码清单 11-2 所示。上面提到其他三个类的处理流程可以参考该案例自行研究学习。

代码清单 11-2 FlowNodeHistoryParseHandler.java

```

1 ActivityInstanceEndHandler ACTIVITI_INSTANCE_END_LISTENER = new ActivityInstanceEndHandler();
2 ActivityInstanceStartHandler ACTIVITI_INSTANCE_START_LISTENER = new
3 ActivityInstanceStartHandler();
4 Set supportedElementClasses = new HashSet();
5 static {
6     supportedElementClasses.add(EndEvent.class); //...省略其他添加操作
7 }
8 public Set<Class<? extends BaseElement>> getHandledTypes() {
9     return supportedElementClasses;
10 }
11 public void parse(BpmnParse bpmnParse, BaseElement element) {
12     ActivityImpl activity = bpmnParse.getCurrentScope().findActivity(element.getId());
13     if(element instanceof BoundaryEvent) {
14         activity.addExecutionListener("end", ACTIVITI_INSTANCE_START_LISTENER, 0);
15         activity.addExecutionListener("end", ACTIVITI_INSTANCE_END_LISTENER, 1);
16     } else {
17         activity.addExecutionListener("start", ACTIVITI_INSTANCE_START_LISTENER, 0);
18         activity.addExecutionListener("end", ACTIVITI_INSTANCE_END_LISTENER);
19     }
20 }

```

在上述代码中,第 1 行实例化 ActivityInstanceEndHandler 类,该类负责监听流程三大要素的结束通知操作,举个例子,就拿任务完成操作来说,任务完成之后,会触发任务节点中事件类型为 end 的内置任务记录监听器中的业务逻辑,也就是当前类 ActivityInstanceEndHandler 中的 notify 方法,notify 方法会更新当前任务节点对应的历史数据,需要更新的属性有如下三个:任务结束时间、任务节点结束的原因(非必须)、节点持续的时间(结束时间-开始时间)。这里可能会有疑问,为何需要更新以上所述的三个属性值,其他属性难道不需要修改或更新吗?姑且留下一个悬念,稍后加以说明。

第 2~3 行实例化 ActivityInstanceStartHandler 类,该类负责监听流程三大要素的开始通知操作,该操作下达的指令恰恰与 ActivityInstanceEndHandler 类完全相反,该操作主要是将任务节点已知的基本信息插入到历史环节表,学习到这里,应该很清楚流程引擎将历史数据插入到数据库的时机。

第 4 行声明了 `supportedElementClasses` 集合, 该集合主要用于存储可以用来动态添加内置记录监听器的 `BaseElement` 实例对象, 并在第 5~7 行进行初始化, 开发人员也可以通过操作该集合添加自定义的内置记录监听器(只有存在于该集合中的 `BaseElement` 实例对象才可以被用来自动添加内置记录监听器)。

第 8 行提供了获取集合元素的方法, 以方便其他模块可以获取到 `supportedElementClasses` 集合的内容。

第 11 行定义 `parse` 方法, 该方法首先根据 `element` 对象的 `id` 值获取该对象对应的流程虚拟机实例对象 `ActivityImpl`, 然后第 13~19 行根据 `ActivityImpl` 实例对象的类型(元素类型)进行区分处理, 其中第 13 行如果对象类型为 `BoundaryEvent` 则为其添加结束记录监听器, 否则添加开始和结束类型的监听器, 监听器的事件类型决定了历史归档记录操作的先后顺序, 可以参考上文的讲解。

经过上面一系列的讲解可以得知, 解析 `supportedElementClasses` 集合中 `BaseElement` 实例对象时, 只要执行 `FlowNodeHistoryParseHandler` 类中的 `parse` 方法就可以轻松地将 `ActivityInstanceEndHandler` 和 `ActivityInstanceStartHandler` 两个记录监听器自动添加到 `BaseElement` 实例对象中, 由衷佩服 Activiti 这样精妙的设计。

11.3.2 初始化历史解析器

上文讲解到只有存在于历史解析器中的 `supportedElementClasses` 集合的 `BaseElement` 实例对象才可以动态添加内置记录监听器, 那么这些历史解析器是如何添加到对象解析器集合中的呢? 第 10.3.1 节中讲解了对对象解析器集合的初始化过程, 该过程调用了 `BpmnParseHandlers` 类的 `addHandlers` 方法进行工作, `addHandlers` 方法的实现如代码清单 11-3 所示。

代码清单 11-3 BpmnParseHandlers.java

```

1  protected Map<Class<? extends BaseElement>, List<BpmnParseHandler>> parseHandlers;
2  public void addHandlers(List<BpmnParseHandler> bpmnParseHandlers) {
3      for (BpmnParseHandler bpmnParseHandler : bpmnParseHandlers) {
4          addHandler(bpmnParseHandler);
5      }
6  }
7  public void addHandler(BpmnParseHandler bpmnParseHandler) {
8      for (Class<? extends BaseElement> type : bpmnParseHandler.getHandledTypes()) {
9          List<BpmnParseHandler> handlers = parseHandlers.get(type);
10         if (handlers == null) {
11             handlers = new ArrayList<BpmnParseHandler>();
12             parseHandlers.put(type, handlers);
13         }
14         handlers.add(bpmnParseHandler);
15     }
16 }

```

在上述代码中, 第 3~5 行遍历 `bpmnParseHandlers` 集合, 然后调用 `addHandler` 方法添

加对象解析器, addHandler 方法首先遍历 bpmnParseHandler 对象的 getHandledTypes 方法的返回值(对应流程文档元素解析后转化的 Activiti 内部表示类 BaseElement), 然后再遍历查找 BaseElement 实例对象所有的解析器集合 handlers, 并最终将 bpmnParseHandler 值添加到 parseHandlers 集合中。

handlers 集合为 List 数据结构, 由此可知同一个 BaseElement 实例可以存在多个对象解析器对其进行解析, 当然也存在执行先后顺序。parseHandlers 集合为 Map 数据结构, 这样程序可以很方便的查询到该对象的所有解析器集合。对于需要动态添加内置记录监听器的 BaseElement 实例来说, 经过上述步骤之后该实例对象对应的历史解析器已经被成功添加, 这样解析该实例对象时, 就可以取出该实例的所有对象解析器, 并循环遍历 parseHandlers 集合进行对象的解析工作。

对于 FlowNodeHistoryParseHandler 类来说, getHandledTypes 方法的返回值正是该类静态代码块中初始化的集合 supportedElementClasses。由于所有的流程元素对象都有可能添加内置记录监听器, 因此没有必要单独为每一个元素添加一个历史解析器, 只需要在 FlowNodeHistoryParseHandler 类中进行统一添加和维护即可, 试想一下如果为每一个 BaseElement 实例对象单独定义一个历史解析器, 那么工作量会非常大, 更致命的问题是导致代码分散, 不容易后期的维护和扩展。

11.3.3 历史节点结束通知

上面提到 FlowNodeHistoryParseHandler 类分别实例化了 ActivityInstanceStartHandler、ActivityInstanceEndHandler 类, 这两个类均实现了 ExecutionListener 接口, 首先分析 ActivityInstanceEndHandler 类, 该类的定义如代码清单 11-4 所示。

代码清单 11-4 ActivityInstanceEndHandler.java

```
1 public void notify(DelegateExecution execution) {
2     Context.getCommandContext().getHistoryManager().recordActivityEnd((ExecutionEntity)
3     execution);
4 }
```

在上述代码中, notify 方法仅仅是委托历史管理器类 HistoryManager 中的活动结束通知方法 recordActivityEnd 进行工作, 该方法的定义如代码清单 11-5 所示。

代码清单 11-5 DefaultHistoryManager.java

```
1 public void recordActivityEnd(ExecutionEntity executionEntity) {
2     if(isHistoryLevelAtLeast(HistoryLevel.ACTIVITY)) { //判断历史数据的归档级别
3         //查询历史数据是否存在
4         HistoricActivityInstanceEntity historicActivityInstance =
5             findActivityInstance(executionEntity);
6         if (historicActivityInstance != null) {
7             endHistoricActivityInstance(historicActivityInstance);
8             //如果历史数据存在, 再开始更新操作
9         }
```



```

10 }
11 Protected void endHistoricActivityInstance(HistoricActivityInstanceEntity
12 historicActivityInstance) {
13     historicActivityInstance.markEnded(null);
14     ...//转发 HISTORIC_ACTIVITY_INSTANCE_ENDED 事件
15 }

```

将 recordActivityEnd 方法的处理逻辑总结如下。

(1) 第 2 行判断归档级别。

归档级别值控制了是否需要记录元素对应的历史环节数据,只有在 activity 之上的级别才可以进行历史数据入库操作。关于历史归档级别稍后进行详解。

(2) 查找历史实例。

第 4~5 行调用 findActivityInstance 方法查找历史流程实例,由于篇幅有限,本书不过多讲解。第 6 行如果 historicActivityInstance 不为空,则第 7 行调用 endHistoricActivityInstance 方法进行处理。

(3) 结束历史实例。

endHistoricActivityInstance 方法主要用于结束历史流程实例,该方法主要分两步完成:首先第委托 markEnded 方法记录结束工作,然后转发 HISTORIC_ACTIVITY_INSTANCE_ENDED 事件。

11.3.4 控制归档历史数据级别

通过上述步骤(1)可知,只有配置的历史归档级别值在 activity 之上,历史环节数据才会进行入库操作,下面分析历史归档的定义以及使用场景,如代码清单 11-6 所示。

代码清单 11-6 activiti.cfg.xml 和 HistoryLevel.java

```

1 activiti.cfg.xml 配置:
2 <beanid = "processEngineConfiguration"
3 class = "org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
4     <property name = "history" value = "audit" />
5 </bean>
6 HistoryLevel.java
7 NONE("none"),ACTIVITY("activity"),AUDIT("audit"),FULL("full");

```

在上述代码中,第 4 行中的 history 开关属性用于配置历史归档级别,可配置的值必须与第 7 行定义的枚举值对应,可以配置的历史级别如下。

(1) none。

所有的历史归档数据不进行入库记录,因此该级别对于流程实例运转来说性能最高,但由于不涉及历史表的操作,客户端一旦配置该级别,则无法查询流程实例的运转轨迹以及细节信息,因此强烈不推荐使用该方式。

(2) activity。

归档所有流程实例以及活动实例,不归档流程细节,例如历史任务节点。

(3) audit。

默认级别,归档所有流程实例、活动实例以及提交的表单属性,所有与用户交互的数据(表单)都是可以跟踪并统计。

(4) full。

历史数据归档的最高级别。该级别记录所有的历史数据,因此流程实例运转时也是最慢的,该级别除了保留 audit 级别的所有信息之外还要保存类似流程变量以及其他可能需要的历史数据。

历史归档级别的优先级从高到低分别为 full、audit、activity、none。

扩展

归档历史数据的处理可以参考 DefaultHistoryManager 类的相关实现。

11.3.5 更新历史数据

了解了历史数据的归档配置以及使用场景之后,下面分析 historicActivityInstance.markEnded(null)方法的处理逻辑,该方法的相关实现如代码清单 11-7 所示。

代码清单 11-7 HistoricScopeInstanceEntity.java

```
1 public void markEnded(String deleteReason) {
2     this.deleteReason = deleteReason;
3     this.endTime = Context.getProcessEngineConfiguration().getClock().getCurrentTime();
4     this.durationInMillis = endTime.getTime() - startTime.getTime();
5 }
```

在上述代码中,第 2~4 行仅仅是填充 HistoricScopeInstanceEntity 抽象类中 deleteReason、endTime、durationInMillis 三个属性值而已,并没有发现任何操作历史表的踪迹,不妨换一个角度,上文多次提到操作历史表,历史表有哪些呢? HistoricScopeInstanceEntity 是一个抽象类并且实现了 PersistentObject 接口,该类的架构如图 11-4 所示。

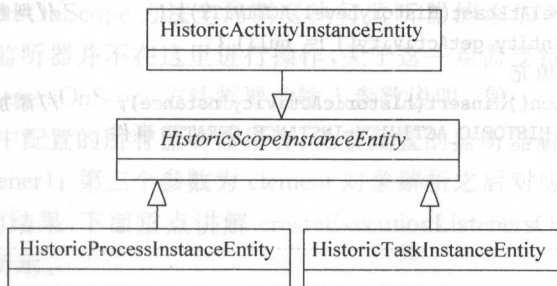


图 11-4 HistoricScopeInstanceEntity 类的架构图

(1) HistoricActivityInstanceEntity 类: 该类为 ACT_HI_ACTINST 表对应的实体类。

(2) HistoricTaskInstanceEntity 类: 该类为 ACT_HI_TASKINST 表对应的实体类。

(3) HistoricProcessInstanceEntity 类: 该类为 ACT_HI_PROCINST 表对应的实体类。

因为图 11-4 中有很多相同的字段,所以将图 11-4 对应实体类中的通用属性封装到父

类 `HistoricScopeInstanceEntity` 中,并让该类实现 `PersistentObject` 接口,该接口定义了 `getPersistentState` 方法,该方法返回 `Object` 类型,该 `Object` 实例对象中的属性值是否在会话缓存中发生变化,决定了以上三个类的实例对象是否需要更新数据库表。`HistoricScopeInstanceEntity` 类中的 `markEnded` 方法可以对以上三个类中 `getPersistentState` 方法中的属性值进行修改操作,该操作均可以在以上三个类中的 `getPersistentState` 方法中发现痕迹。换言之,只要以上三个类中 `getPersistentState` 方法中定义的任意一个属性值发生了变化,则命令上下文 `CommandContext` 中的 `close` 方法执行刷新会话缓存操作的同时会将变化的数据更新到数据库,该处理过程可以参考第 15.11 节。基于上述的讲解可知 `ActivityInstanceEndHandler` 类主要是为了完成对历史数据的更新操作。

11.3.6 历史节点开始通知

接下来,分析 `ActivityInstanceStartHandler` 类,该类的定义如代码清单 11-8 所示。

代码清单 11-8 `ActivityInstanceStartHandler.java`

```
1 public class ActivityInstanceStartHandler implements ExecutionListener {
2     public void notify(DelegateExecution execution) {
3         Context.getCommandContext().getHistoryManager()
4             .recordActivityStart((ExecutionEntity) execution);
5     }
6 }
```

`ActivityInstanceStartHandler` 类实现了 `ExecutionListener` 接口并对该接口中的 `notify` 方法进行了实现, `notify` 方法首先获取历史管理器 `HistoryManager`, 然后调用该管理器中的记录活动开始方法 `recordActivityStart`, `recordActivityStart` 方法的相关实现如代码清单 11-9 所示。

代码清单 11-9 `DefaultHistoryManager.java`

```
1 public void recordActivityStart(ExecutionEntity executionEntity) {
2     if(isHistoryLevelAtLeast(HistoryLevel.ACTIVITY)) {           //判断历史数据的归档级别
3         if(executionEntity.getActivity() != null) {
4             ...//省略属性填充
5             getDbSqlSession().insert(historicActivityInstance);    //添加到缓存
6             ...//省略转发 HISTORIC_ACTIVITY_INSTANCE_CREATED 事件
7         }
8     }
9 }
```

从代码上来看,活动记录操作经历了以下几个过程。

(1) 第 2 行判断历史归档级别。

只有历史级别值在 `activity` 之上才可以进行历史数据的入库操作。

(2) 封装 `HistoricActivityInstanceEntity` 实例对象并为其填充属性。

`HistoricActivityInstanceEntity` 类为历史实例表 `ACT_HI_PROCINST` 对应的实体类,因此 `HistoricActivityInstanceEntity` 类中定义的属性与 `ACT_HI_PROCINST` 表中的字段

一一对应。

(3) 添加到缓存中。

第 5 行直接调用 `getDbSqlSession().insert` 方法将 `historicActivityInstance` 对象添加到会话缓存中。

(4) 转发 `HISTORIC_ACTIVITY_INSTANCE_CREATED` 事件。

通过 `recordActivityStart` 方法的执行逻辑可知 `ActivityInstanceStartHandler` 类负责将历史数据添加到会话缓存中,引擎最终会将会话缓存中的数据刷新到数据库。

11.4 注入执行监听器

内置记录监听器的相关内容已经讲解完毕,在讲解内置记录监听器如何被引擎触发之前,再次结合第 10.3.1 节分析监听器是如何添加到 `BaseElement` 实例对象中的,首先分析 `AbstractFlowNodeBpmnParseHandler` 类,该类作为对象解析器的模板类存在,该类中的 `parse` 方法主要完成两大功能:①调用父类方法对 `BaseElement` 实例对象进行解析;②为 `BaseElement` 实例对象的解析结果(`ActivityImpl` 实例对象)添加执行监听器,该类的核心定义如代码清单 11-10 所示。

代码清单 11-10 AbstractFlowNodeBpmnParseHandler.java

```
1 public void parse(BpmnParse bpmnParse, BaseElement element) {  
2     //调用父类进行 element 的解析  
3     super.parse(bpmnParse, element);  
4     //调用父类为 element 对象添加执行监听器  
5     createExecutionListenersOnScope(bpmnParse, ((FlowNode)element).getExecutionListeners(),  
6     findActivity(bpmnParse, element.getId()));  
7 }
```

在上述代码中,第 3 行委托父类 `AbstractBpmnParseHandler` 将 `element` 转化为 `ActivityImpl` 实例对象;因为所有的流程元素都可以配置执行监听器,所以第 5~6 行调用 `createExecutionListenersOnScope` 方法全局调度执行监听器的添加工作,该操作只针对执行监听器的添加,任务监听器并不在这里进行操作,关于这一点需要特别注意。

`createExecutionListenersOnScope` 方法需要的输入参数说明:第一个参数为 `bpmnParse` 对象;第二个参数为元素中配置的所有监听器集合(元素配置的监听器解析之后转化为引擎的内部表示类 `ActivitiListener`);第三个参数为 `element` 对象解析之后对应的 `ActivityImpl` 实例,对应第 3 行操作之后的结果,下面重点讲解 `createExecutionListenersOnScope` 方法的处理逻辑,如代码清单 11-11 所示。

代码清单 11-11 AbstractBpmnParseHandler.java

```
1 void createExecutionListenersOnScope(BpmnParse bpmnParse,  
2 List<ActivitiListener> activitiListenerList, ScopeImpl scope) {  
3     for (ActivitiListener activitiListener : activitiListenerList) { //遍历 activitiListenerList  
4         scope.addExecutionListener(activitiListener.getEvent(),  
5         createExecutionListener(bpmnParse, activitiListener));
```



```

6     }
7 }
8 protected ExecutionListener createExecutionListener(BpmnParse bpmnParse, ActivitiListener
9 activitiListener) {
10     ExecutionListener executionListener = null;
11     //根据执行监听器的类型,实例化不同的 ExecutionListener 子类
12     if ("class".equalsIgnoreCase(activitiListener.getImplementationType())) {
13         executionListener = bpmnParse.getListenerFactory().
14         createClassDelegateExecutionListener(activitiListener);
15     } else if ("expression".equalsIgnoreCase(activitiListener.getImplementationType())) {
16         executionListener = bpmnParse.getListenerFactory().
17         createExpressionExecutionListener(activitiListener);
18     } else if ("delegateExpression".equalsIgnoreCase(activitiListener.getImplementationType())
19     )) {
20         executionListener = bpmnParse.getListenerFactory().
21         createDelegateExpressionExecutionListener(activitiListener);
22     }
23     return executionListener;
24 }

```

将上面代码的执行逻辑梳理总结如下。

(1) 第 3~6 行循环遍历元素的所有监听器集合 `activitiListenerList`。

不管是执行监听器还是任务监听器,元素解析完后都会使用 `ActivitiListener` 实例对象进行表示。第 3 行遍历 `activitiListenerList` 参数值,然后委托 `ScopeImpl` 类中的 `addExecutionListener` 方法添加执行监听器,该方法需要两个输入参数,第一个参数为监听器的事件类型,第二个参数为 `ExecutionListener` 类型,通过该步骤之后 `scope` 对象中已经存在了执行监听器。`ScopeImpl` 类是 `ActivityImpl` 类的父类,这样后续流程实例运转时,如果探测到 `ActivityImpl` 实例对象中有执行监听器,则直接触发执行监听器中的方法,稍后即看到。

(2) 创建执行监听器。

为何需要将 `ActivitiListener` 实例对象转化为 `ExecutionListener` 实例对象呢? 其实不难理解,因为所有的执行监听器均需要实现 `ExecutionListener` 接口,在这里对其进行转化的目的,是为了方便对所有的执行监听器进行全局统一调度,而无需关心其内部的具体实现细节,这种为典型的面向接口编程思想。虽然到目前为止,还没讲解任务监听器的解析转换注入工作,但是根据这里的处理逻辑也可以大胆推测猜想,任务监听器肯定是将 `ActivitiListener` 实例对象转化为 `TaskListener` 实例对象统一调度,因为所有的任务监听器均需要实现 `TaskListener` 接口。通过上文中 `createExecutionListener` 方法的处理逻辑可以看出, `class` 方式创建的执行监听器使用 `ClassDelegate` 类进行封装, `expression` 方式创建的执行监听器使用 `ExpressionExecutionListener` 类进行封装, `delegateExpression` 方式创建的执行监听器使用 `DelegateExpressionExecutionListener` 类进行封装。

`createExecutionListener` 方法主要用于创建执行监听器,因为执行监听器或者任务监听器均可以通过 `class`、`expression` 或者 `delegateExpression` 方式进行定义,所以在执行监听器的创建过程中会根据其创建方式委托给 `DefaultListenerFactory` 类中不同的方法进行处

理,本节以 class 创建方式为例,详细讲解执行监听器的创建过程,该方法的相关定义如代码清单 11-12 所示。

代码清单 11-12 DefaultListenerFactory.java

```
1 ExecutionListener createClassDelegateExecutionListener(ActivitiListener activitiListener)
{
2     return new ClassDelegate(activitiListener.getImplementation(),
3         createFieldDeclarations(activitiListener.getFieldExtensions()));
4 }
```

createClassDelegateExecutionListener 方法的执行逻辑非常简洁,总结如下。

(1) 根据 activitiListener 获取具体的执行监听器实例对象。

(2) 为获取的监听器实例对象填充属性。

createFieldDeclarations 方法的主要工作就是将监听器中 FieldExtension 类型的对象,转化为 FieldDeclaration 类型对象,该方法的执行逻辑如代码清单 11-13 所示。

代码清单 11-13 AbstractBehaviorFactory.java

```
1 public List<FieldDeclaration> createFieldDeclarations(List<FieldExtension> fieldList) {
2     List<FieldDeclaration> fieldDeclarations = new ArrayList<FieldDeclaration>();
3     for (FieldExtension fieldExtension : fieldList) { //遍历 fieldList 集合
4         FieldDeclaration fieldDeclaration = null;
5         if (StringUtils.isEmpty(fieldExtension.getExpression())) {
6             fieldDeclaration = new FieldDeclaration(fieldExtension.getFieldName(), Expression.class,
7                 getName(), expressionManager.createExpression(fieldExtension.getExpression()));
8         } else {
9             fieldDeclaration = new FieldDeclaration(fieldExtension.getFieldName(), Expression.
10                class,
11                getName(), new FixedValue(fieldExtension.getStringValue()));
12            fieldDeclarations.add(fieldDeclaration);
13        }
14    }
15    return fieldDeclarations;
16 }
```

因为监听器中的表达式属性解析之后被封装为 FieldExtension 实例,而流程虚拟机运转过程中需要的属性类型为 FieldDeclaration 实例,所以这里需要进行统一转换,转换过程比较简单,第 3~13 行遍历 fieldList 集合,如果发现 fieldExtension 对象中的属性配置有表达式则使用第 6~7 行实例化 FieldDeclaration 类;否则第 9~10 行实例化 FieldDeclaration 类,两者唯一的区别就是第 7 行使用表达式,第 10 行使用 FixedValue 类。

(3) 根据执行监听器和该类对应的属性值实例化 ClassDelegate 类。

实例化 ClassDelegate 类的过程非常简单,需要传递的参数有具体的执行监听器实例对象以及该类中定义的属性信息值。至此 class 方式创建执行监听器的过程已经分析完毕。

建议

expression 和 delegateExpression 方式创建执行监听器的原理类似,可以结合上述的讲解自行学习。

11.5 注入任务监听器

因为所有的流程元素都可以添加不同类型的执行监听器,所以上文中看到了 `createExecutionListenersOnScope` 方法全局调度执行监听器的添加工作,由于任务监听器只可以在任务节点中进行配置使用,所以没必要将任务监听器的解析添加工作放到公共方法中,只需要解析任务节点的时候处理即可。任务节点解析类 `UserTaskParseHandler` 的实现逻辑如代码清单 11-14 所示。

代码清单 11-14 `UserTaskParseHandler.java` 和 `TaskDefinition.java`

```

1 UserTaskParseHandler.java
2 public TaskDefinition parseTaskDefinition(BpmnParse bpmnParse, UserTask userTask, String
3 taskDefinitionKey, ProcessDefinitionEntity processDefinition) {
4     ...//省略一系列的属性设置
5     for (ActivitiListener taskListener : userTask.getTaskListeners()) {
6         taskDefinition.addTaskListener(taskListener.getEvent(),
7             createTaskListener(bpmnParse, taskListener, userTask.getId()));
8     }
9     return taskDefinition;
10 }
11 TaskDefinition.java
12 public void addTaskListener(String eventName, TaskListener taskListener) {
13     if ("all".equals(eventName)) {
14         this.addTaskListener("create", taskListener);
15         this.addTaskListener("assignment", taskListener);
16         this.addTaskListener("complete", taskListener);
17         this.addTaskListener("delete", taskListener);
18     } else {
19         List<TaskListener> taskEventListeners = taskListeners.get(eventName);
20         if (taskEventListeners == null) {
21             taskEventListeners = new ArrayList<TaskListener>();
22             taskListeners.put(eventName, taskEventListeners);
23         }
24         taskEventListeners.add(taskListener);
25 }

```

任务监听器注入其作用的 `UserTask` 实例对象的过程中, `parseTaskDefinition` 方法将任务节点对象解析之后的结果封装到 `TaskDefinition` 实例对象中。

任务监听器中的 `all` 事件类型在 `TaskDefinition` 类的 `addTaskListener` 方法中进行判断处理, `createTaskListener` 方法的执行逻辑与上文讲解的执行监听器的创建过程类似,唯一的区别是这里需要创建 `TaskListener` 实例对象而后者需要创建 `ExecutionListener` 实例对象。

任务监听器的创建方式也分为如下三种: `class` 方式创建的任务监听器使用 `ClassDelegate` 类进行封装, `expression` 方式创建的任务监听器使用 `ExpressionTaskListener` 类进行封装, `delegateExpression` 方式创建的任务监听器使用 `DelegateExpressionTaskListener` 类

进行封装。为何用户配置的执行监听器或者任务监听器需要使用不同的类进行封装呢？这就涉及了引擎触发监听器的原理，接下来重点讲解。

11.6 触发执行监听器

上文讲解了监听器注入其作用节点对象的整个过程，接下来详细分析引擎是如何触发监听器的。需要知道一点，所有执行监听器的调用均通过原子类 `AbstractEventAtomicOperation` 触发，可以参考流程第 13 章，暂且将关注点放置在该原子类中 `execute` 方法的处理逻辑中，如代码清单 11-15 所示。

代码清单 11-15 `AbstractEventAtomicOperation.java`

```
1 public void execute(InterpretableExecution execution){
2     ScopeImpl scope = getScope(execution);           //获取 scope 对象
3     //获取同一种事件类型的监听器集合
4     List<ExecutionListener> executionListeners = scope.getExecutionListeners(getEventName());
5     int executionListenerIndex = execution.getExecutionListenerIndex();
6     if (executionListeners.size() > executionListenerIndex) {
7         execution.setEventName(getEventName());
8         execution.setEventSource(scope);
9         ExecutionListener listener = executionListeners.get(executionListenerIndex);
10        listener.notify(execution);                    //触发监听器中的 notify 方法
11        ...//省略原子类运转过程
12 }
```

从上面的代码中可以看出，执行监听器的触发逻辑非常简单，可以将其梳理总结如下。

(1) 第 2 行获取 `ScopeImpl` 实例对象。

因为流程实例运转时，需要知道当前所处理的节点，所以第 2 行直接根据 `execution` 从流程虚拟机中获取当前节点的信息值。

(2) 第 4 行根据事件类型获取元素中配置的执行监听器集合 `executionListeners`。

(3) 循环遍历监听器集合。

因为同一个元素的同一个事件类型的执行监听器可以存在多个，所以需要根据事件类型循环遍历集合中的监听器并执行。

(4) 封装 `execution`。

触发执行监听器时需要将 `InterpretableExecution (ExecutionEntity)` 实例对象作为 `notify` 方法的输入参数进行传递，传递该参数主要是为了方便开发人员直接通过 `notify` 方法中的 `DelegateExecution` 类型的参数获取需要的信息。

(5) 触发 `notify` 方法。

第 9 行获取元素对应的执行监听器对象 `listener`，第 10 行触发执行监听器的 `notify` 方法。

11.6.1 class 方式调度

试想一下如果程序需要执行用户配置的监听器，则首先需要获取该类的实例对象，然后

调用该实例对象中的 `notify` 方法,由于用户创建执行监听器的方式不同,最终封装的实例对象也不同,内置记录监听器则由引擎直接触发,关于这一点一定要牢记。`class` 方式创建的执行监听器封装为 `ClassDelegate` 实例对象,所以暂且将关注点放到 `ClassDelegate` 类中 `notify(DelegateExecution execution)` 方法的处理逻辑中,如代码清单 11-16 所示。

代码清单 11-16 ClassDelegate.java

```

1 protected ExecutionListener executionListenerInstance;
2 public void notify(DelegateExecution execution) throws Exception {
3     if (executionListenerInstance == null) {
4         executionListenerInstance = getExecutionListenerInstance();
5     }
6     Context.getProcessEngineConfiguration().getDelegateInterceptor().
7     .handleInvocation(new ExecutionListenerInvocation(executionListenerInstance, execution));
8 }

```

将其处理流程梳理总结如下。

(1) 第 3 行对 `executionListenerInstance` 对象进行非空校验,如果该对象为空,则第 4 行实例化需要执行的监听器。

(2) 第 6~8 行最终会调用 `ExecutionListenerInvocation` 实例对象中的 `invoke` 方法, `invoke` 方法最终会触发执行监听器实例对象中的 `notify` 方法。

`getExecutionListenerInstance` 方法主要用来创建执行监听器实例对象,该方法的定义如代码清单 11-17 所示。

代码清单 11-17 ClassDelegate.java

```

1 protected ExecutionListener getExecutionListenerInstance(){
2     Object delegateInstance = instantiateDelegate(className, fieldDeclarations);
3     if (delegateInstance instanceof ExecutionListener) {
4         return (ExecutionListener) delegateInstance;
5     } else if (delegateInstance instanceof JavaDelegate) {
6         return new ServiceTaskJavaDelegateActivityBehavior(((JavaDelegate) delegateInstance));
7     } else {
8         throw new ActivitiIllegalArgumentException(delegateInstance.getClass().getName());
9     }
10 }

```

在上述代码中,首先第 2 行根据类名(`className`)和“属性”(`fieldDeclarations`)两个值实例化执行监听器,然后第 3~9 行根据该实例对象的类型进行转换,如果执行监听器没有实现 `ExecutionListener` 或者 `JavaDelegate` 接口,则第 8 行程序直接报错,该操作隐含透露一个信息,用户配置的执行监听器可以实现 `ExecutionListener` 接口,也可以实现 `JavaDelegate` 接口,在这里进行类型转化主要是为了方便对具体的监听器进行全局统一调度,并约束执行监听器的实现类必须实现以上两个接口中的任意一个。`instantiateDelegate` 方法的相关实现如代码清单 11-18 所示。

代码清单 11-18 ClassDelegate.java

```

1  protected Object instantiateDelegate(String className, List fieldDeclarations) {
2      return ClassDelegate.defaultInstantiateDelegate(className, fieldDeclarations);
3  }
4  static Object defaultInstantiateDelegate(String className, List fieldDeclarations) {
5      Object object = ReflectUtil.instantiate(className); //通过反射实例化类
6      applyFieldDeclaration(fieldDeclarations, object); //对 object 对象进行属性填充
7      return object;
8  }
9  public static void applyFieldDeclaration(List fieldDeclarations, Object target, boolean
10 throwExceptionOnMissingField) {
11      if(fieldDeclarations != null) { //如果 fieldDeclarations 不为空
12          for(FieldDeclaration declaration : fieldDeclarations){
13              applyFieldDeclaration(declaration, target, throwExceptionOnMissingField);
14          }
15      }
16  }
17  public static void applyFieldDeclaration(FieldDeclaration declaration, Object target, boolean
18 throwExceptionOnMissingField) {
19      Method setterMethod = ReflectUtil.getSetter(declaration.getName(), //获取方法
20 target.getClass(), declaration.getValue().getClass());
21      if(setterMethod != null) {
22          setterMethod.invoke(target, declaration.getValue());
23          //调用 setterMethod 方法进行属性赋值
24      } else {
25          Field field = ReflectUtil.getField(declaration.getName(), target); //获取属性信息
26          if(field == null) {
27              if (throwExceptionOnMissingField) {
28                  throw new ActivitiIllegalArgumentException("Field definition ");
29              } else {
30                  return;
31              }
32          } if(!fieldTypeCompatible(declaration, field)) {
33              throw new ActivitiIllegalArgumentException("Incompatible type ");
34          } ReflectUtil.setField(field, target, declaration.getValue()); //设置属性值
35      }
36  }

```

仅从代码量上就能看出执行监听器的加载和实例化过程相当复杂,其中涉及了各种各样的考虑,处理逻辑大致总结如下。

- (1) 第2行委托 ClassDelegate 类中的 defaultInstantiateDelegate 方法进行下一步处理。
- (2) 第5行使用反射方式实例化执行监听器,具体实现如代码清单 11-19 所示。

代码清单 11-19 ReflectUtil.java

```

1  public static Object instantiate(String className) {
2      Class<?> clazz = loadClass(className); //使用类加载器根据 className 获取类的信息
3      return clazz.newInstance(); //实例化类的对象
4  }

```

这里需要明白一个问题,引擎使用反射技术创建执行监听器实例对象,因此如果期望通过 class 方式定义的执行监听器获取 Spring 容器中的对象,很显然该方式是做不到的,因为使用反射方式创建监听器的过程中是不会获取 Spring 容器中的实例对象的。

(3) 第 6 行委托 applyFieldDeclaration 方法工作。其中,第 9~10 行定义的方法的工作就是将执行监听器中的表达式属性信息值填充到监听器实例对象中,因为执行监听器中可以配置多个表达式“属性”,因此第 12~14 行需要循环遍历属性集合并注入。

第 17~18 行定义的方法负责为执行监听器实例对象填充属性值,处理逻辑比较简单:首先根据类中的属性名称获取该属性对应的 setter 方法,比如属性名为 sharenui 则对应的赋值方法为 setSharenui,如果找到了 setSharenui 方法,则执行第 22 行开始进行处理;如果该属性没有 setter 方法,则第 24 行直接通过反射方式获取 field 对象,如果 field 对象为空,第 26 行根据 throwExceptionOnMissingField 参数值进行处理,如果该参数值为 true,第 27 行程序报错,否则第 29 行直接返回。

上述一系列步骤执行完毕,如果 field 对象不为空,第 31 行调用 fieldTypeCompatible 方法进行处理,如果该方法返回 false,程序直接报错,否则第 34 行使用反射方式设置属性值。

11.6.2 delegateExpression 方式调度

delegateExpression 方式创建执行监听器通常与 Spring 框架结合起来使用,如代码清单 11-20 所示。

代码清单 11-20 delegateExpression 方式创建执行监听器

```

1  流程文档:
2  <extensionElements>
3      <!-- 事件类型为 start 的执行监听器 委托表达式的配置如 delegateExpression 所示 -->
4      <activiti:executionListenablevent = "start" delegateExpression = "${sharenui}"/>
5  </extensionElements>
6  activiti.cfg.xml:
7  <bean id = "sharenui" class = "com.sharenui.chapter11.SharenuiDelegateExpression"></bean>
8  SharenuiDelegateExpression.java
9  public class SharenuiDelegateExpression implements Serializable,JavaDelegate{
10      public void execute(DelegateExecution execution) throws Exception {
11      }
12  }
```

在上述代码中,第 2~5 行通过 delegateExpression 方式定义了事件类型名称为 start 的执行监听器,其中 delegateExpression 属性值为 \${sharenui},sharenui 值可以是流程实例运行时的变量名称,形如 variables.put("sharenui", new SharenuiDelegateExpression()),也可以与 Spring 框架结合起来使用,例如第 7 行定义了一个 id 值为 sharenui 的 SharenuiDelegateExpression 类,第 9 行定义一个类并实现了 JavaDelegate 接口,接下来分析 Activiti 是如何处理该方式定义的执行监听器,DelegateExpressionExecutionListener 类的定义如代码清单 11-21 所示。

代码清单 11-21 DelegateExpressionExecutionListener.java

```

1 public void notify(DelegateExecution execution) throws Exception {
2     Object delegate = expression.getValue(execution);
3     ClassDelegate.applyFieldDeclaration(fieldDeclarations, delegate);
4     if (delegate instanceof ExecutionListener) {
5         Context.getProcessEngineConfiguration().getDelegateInterceptor().handleInvocation(new
6             ExecutionListenerInvocation((ExecutionListener) delegate, execution));
7     } else if (delegate instanceof JavaDelegate) {
8         Context.getProcessEngineConfiguration().getDelegateInterceptor().
9             handleInvocation(new JavaDelegateInvocation((JavaDelegate) delegate, execution));
10    } else {
11        throw new ActivitiIllegalArgumentException( );
12    }
13 }

```

Activiti 引擎对 delegateExpression 方式与 class 方式定义的执行监听器的处理逻辑几乎相同, class 方式通过反射实例化执行监听器, 然后对其进行属性填充, 上述代码中, 第 2 行获取监听器实例对象 delegate, 第 3 行填充 delegate 对象属性, 然后第 4~12 行校验 delegate 对象是否是 ExecutionListener 实例对象或者 JavaDelegate 实例对象并根据 delegate 对象的类型触发相应的监听器。

11.6.3 expression 方式调度

expression 方式定义的监听器比较灵活, 但使用起来比较复杂, 该方式可以直接设置需要调用的类以及类中的方法, 如代码清单 11-22 所示。

代码清单 11-22 expression 方式定义的执行监听器

```

1 Person.java:
2 public class Person {
3     private int a;
4 }
5 activiti.cfg.xml:
6 <bean id="person" class="com.shareniu.chapter11.Person">
7     <property name="a" value="1"></property>
8 </bean>
9 流程文档:
10 <extensionElements>
11     <!-- 事件类型为 start 的执行监听器 表达式的配置如 expression 所示 -->
12     <activiti:executionListener event="start" expression="$ {shareniu.doSomething
13         (execution)}"/>
14     <activiti:executionListener event="start"
15         expression="$ {shareniu.doSomething1(person, 'b', shareniu)}"/>
16 </extensionElements>
17 ShareniuExpress.java:
18 public class ShareniuExpress implements Serializable {
19     public void doSomething(DelegateExecution execution) throws Exception {
20         execution.setVariable("myVar", execution.getVariable("msg"));

```



```

20     }
21     public void doSomething1(Person person, String b, String shareniu) throws Exception {
22     }
23 }
24 App.java:
25 Map<String, Object> variables = new HashMap<String, Object>();
26 variables.put("shareniu", 20);
27 runtimeService.startProcessInstanceId("myProcess:12:110004", variables);

```

在上述代码中,第2行定义了 Person 类,并通过第6~8行将其交给 Spring 容器进行管理,第10~15行通过 expression 属性分别定义了两个执行监听器,其中 expression 属性值中的 shareniu 表示定义的执行监听器,可以使用变量的方式,也可以与 Spring 框架结合使用。doSomething 和 doSomething1 两个方法分别对应 ShareniuExpress 类中的第18行和第21行定义的方法,当然方法需要的输入参数名称可以是任意值,只要变量的类型和个数与第12行和第14行定义的一致即可。Activiti 为了方便开发人员使用,提供了如下几个内置变量。

(1) execution: DelegateExecution 类型,保存了流程执行的相关信息。

(2) task: DelegateTask 类型,保存了与当前任务相关信息(适用于任务节点)。

以上两个类型的变量是系统内置的。需要说明一点,DelegateTask 类型的变量,仅可以在任务监听器中进行配置和使用。第21行方法中定义的两个参数说明: person 参数为 Person 实例对象(交给 Spring 容器进行管理),b 参数为 String 类型的固定值,shareniu 参数为 String 类型,可以通过变量的方式进行设置形如第26行所示。接下来分析 Activiti 对其调用处理的过程,如代码清单 11-23 所示。

代码清单 11-23 ExpressionExecutionListener.java

```

1 public void notify(DelegateExecution execution) throws Exception {
2     expression.getValue(execution);
3 }

```

notify 方法直接通过 expression.getValue(execution) 方法对执行监听器中的方法进行调用,其内部使用 Juel 对其进行处理。

11.6.4 执行监听器触发入口

暂且不考虑拦截器的使用,分析执行监听器的触发逻辑,如代码清单 11-24 所示。

代码清单 11-24 ExecutionListenerInvocation.java

```

1 protected void invoke() throws Exception {
2     executionListenerInstance.notify(execution);
3 }

```

该方法的处理逻辑比较简单,直接触发执行监听器实例对象 executionListenerInstance 中的 notify 方法,并将 execution 对象作为参数进行传递。

扩展

关于实现 JavaDelegate 接口的执行监听器的调用过程可以参考 JavaDelegateInvocation 类中的相关实现。

11.7 触发任务监听器

上文详细讲解了执行监听器被触发的整个过程,因为任务监听器的设计思路与执行监听器基本相同,只是实例化的类不同而已,所以接下来快速了解一下任务节点中不同事件的监听器的调用先后顺序。需要明确一点,当流程实例运转到任务节点时,会执行任务节点的行为类 UserTaskActivityBehavior 并分配任务节点的处理人以及任务节点的创建信息,整个过程如代码清单 11-25 所示。

代码清单 11-25 UserTaskActivityBehavior.java 和 TaskEntity.java

```

1  UserTaskActivityBehavior.java
2  public void execute(ActivityExecution execution) throws Exception {
3      ...//省略非任务监听器执行代码
4      handleAssignments(activeAssigneeExpression, activeOwnerExpression,
5          activeCandidateUserExpressions, activeCandidateGroupExpressions, task, execution);
6      task.fireEvent(TaskListener.EVENTNAME_CREATE);
7  }
8  protected void handleAssignments(Expression assigneeExpression, Expression ownerExpression, Set
    <Expression> candidateUserExpressions,
9      Set<Expression> candidateGroupExpressions, TaskEntity task, ActivityExecution execution) {
10     ...// 省略非任务监听器执行代码
11     task.setAssignee(assigneeValue, true, false);
12 }
13 TaskEntity.java
14 public void setAssignee(String assignee, boolean dispatchAssignmentEvent, boolean
15 dispatchUpdateEvent) {
16     ...//省略非任务监听器执行代码
17     if(!StringUtils.equals(initialAssignee, assignee)) {
18         fireEvent(TaskListener.EVENTNAME_ASSIGNMENT);
19         initialAssignee = assignee;
20     }
21 }
22 public void fireEvent(String taskEventName) {
23     TaskDefinition taskDefinition = getTaskDefinition();
24     if (taskDefinition != null) {
25         List<TaskListener> taskEventListeners =
26             getTaskDefinition().getTaskListener(taskEventName);
27         if (taskEventListeners != null) {
28             for (TaskListener taskListener : taskEventListeners) {
29                 //省略非任务监听器执行代码
30                 Context.getProcessEngineConfiguration().getDelegateInterceptor().
31                     handleInvocation(new TaskListenerInvocation(taskListener, (DelegateTask)this));
32             }

```

```

33     }
34     }
35 }

```

暂且将关注点放到任务节点的处理人设置以及创建任务节点的环节上,并尝试将其处理逻辑梳理如下。

(1) 第 4~5 行调用 `handleAssignments` 方法设置了当前任务节点的处理人,第 11 行 `task.setAssignee(assigneeValue, true, false)` 方法内部直接调用 `TaskEntity` 类中的 `setAssignee` 方法进行处理,并触发事件类型为 `assignment` 的任务监听器。

(2) 第 6 行调用 `TaskEntity` 类的 `fireEvent` 方法并触发事件类型为 `create` 的任务监听器。

通过上面的处理步骤可知任务监听器首先触发 `assignment` 类型的任务监听器,然后触发 `create` 类型的任务监听器。需要注意的是,第 30~31 行开始触发任务监听器,任务监听器的触发逻辑与执行监听器的处理逻辑相同,这里不再详细地讲解了。

扩展

必须在流程文档中定义任务处理人,或者在到达该任务之前显式调用 `setAssignee` 设置当前的任务处理人,或者调用 `TaskService` 实例对象中的 `claim` 方法才可以触发 `assignment` 类型的任务监听器,设置任务候选人或者候选组则不会触发 `assignment` 类型的任务监听器。`delete` 类型的任务监听器可以参考 `TaskEntityManager` 类中的 `deleteTask` 方法。

11.8 监听器代理

11.8.1 默认代理类

了解了执行监听器和任务监听器的实例化过程以及调用逻辑之后,再次分析代码清单 11-16 中的 `notify` 方法,该方法的处理过程为:首先从 `Context` 类中获取 `ProcessEngineConfigurationImpl` 实例对象,然后根据该实例对象获取 `delegateInterceptor` 对象,最后根据 `ProcessEngineConfigurationImpl` 实例对象以及执行监听器实例对象实例化 `ExecutionListenerInvocation` 类。看到这里可能会有疑问:既然已经获取到了监听器实例对象,为何不直接触发监听器中的 `notify` 方法呢?因为 Activiti 在触发用户配置监听器中的 `notify` 方法之前,做了一层全局功能架构,即使用代理模式对监听器的访问进行控制,这样设计之后,就可以对所有需要执行的监听器进行拦截,从而控制监听器是否可以执行。在实际项目开发中可以通过该特性对废弃的监听器进行屏蔽。基于此, `DelegateInterceptor` 类应运而生, `DelegateInterceptor` 类的功能架构如图 11-5 所示。

根据图 11-5 的类图可以看出, `DelegateInterceptor` 接口的默认实现类为 `DefaultDelegateInterceptor`,该接口定义了 `handleInvocation` 方法拦截以及调用所有需要执行的监听器实例对象, `handleInvocation` 方法直接委托 `DelegateInvocation` 实例对象中的 `proceed` 方法调用监听器。 `DelegateInvocation` 抽象类中定义了 `proceed` 方法(执行用户自定义监听

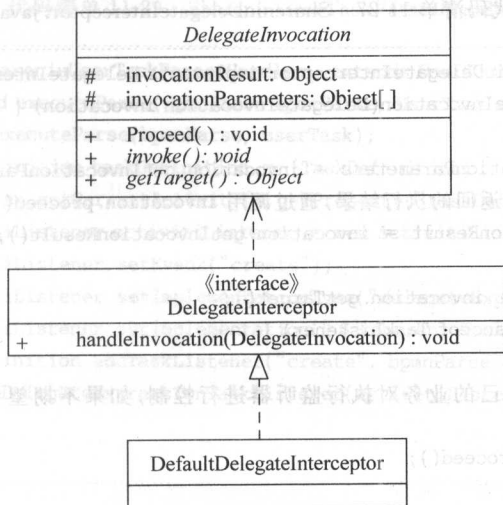


图 11-5 DelegateInterceptor 类的功能架构图

器)、invoke 方法和 getTarget 方法,明白了这样的设计意图之后,再次看一下 ExecutionListenerInvocation 类的实例化过程,如代码清单 11-26 所示。

代码清单 11-26 ExecutionListenerInvocation.java

```
1 public class ExecutionListenerInvocation extends DelegateInvocation {
2     protected final ExecutionListener executionListenerInstance;
3     protected final DelegateExecution execution;
4     protected void invoke() throws Exception {
5         executionListenerInstance.notify(execution);
6     }
7     public Object getTarget() {
8         return executionListenerInstance;
9     }
10 }
```

在上述代码中,invoke 方法直接调用监听器中的 notify 方法,第 2 行和第 3 行定义的两个属性在 ExecutionListenerInvocation 类实例化的同时已经被初始化。至此,有关监听器的触发过程已经讲解完毕。

建议

可以结合该案例查看 DelegateInvocation 类的相关子类进行学习。

11.8.2 自定义代理类

上文讲解了 Activiti 使用代理模式对用户自定义监听器的访问进行拦截控制。下面自定义一个拦截器 ShareniuDelegateInterceptor 类,以此实现拦截功能,该类的详细定义如代码清单 11-27 所示。

代码清单 11-27 ShareniuDelegateInterceptor.java

```

1 public class ShareniuDelegateInterceptor implements DelegateInterceptor {
2     public void handleInvocation(DelegateInvocation invocation) {
3         //调用的参数
4         Object[] invocationParameters = invocation.getInvocationParameters();
5         //调用之后方法返回的执行结果,通过调用 invocation.proceed()之后才有值
6         Object invocationResult = invocation.getInvocationResult();
7         //执行的目标类
8         Object target = invocation.getTarget();
9         if (target instanceof TaskListener) {
10             }else {
11                 //可以根据自己的业务对执行监听器进行控制,如果不期望目标执行则不调用 proceed
12                 invocation.proceed();
13             }
14         }
15     }

```

在上述代码中,定义了 ShareniuDelegateInterceptor 类并实现 DelegateInterceptor 接口,第 12 行中的 invocation.proceed()方法决定了是否可以调用监听器,所以可以根据自身的业务需求对自定义监听器进行访问控制,最后将自定义类注入流程引擎配置类如代码清单 11-28 所示。

代码清单 11-28 activiti.cfg.xml

```

1 <bean id="delegateInterceptor" class="com.shareniu.chapter11.ShareniuDelegateInterceptor"/>
2 <bean id="processEngineConfiguration"
3     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
4     <property name="delegateInterceptor" ref="delegateInterceptor"/></property>
5 </bean>

```

在上述代码中,第 1 行定义了 ShareniuDelegateInterceptor 类,第 4 行通过 delegateInterceptor 开关属性将其注入流程引擎配置类。

11.9 自定义全局任务监听器

在实际项目开发中,定义流程文档时可能会大量使用任务节点,并需要为其添加任务监听器。试想一下,如果流程文档中有成千上百个任务节点,则添加任务监听器的工作量非常大,有没有一种一劳永逸的方法配置任务监听器呢,这也是接下来重点讲解的地方,即全局任务监听器。经过前面的学习可知,任务监听器的添加工作是在 UserTaskParseHandler 类的 executeParse 方法中完成的,所以可以自定义一个类并让其继承 UserTaskParseHandler 类,然后在 executeParse 方法中添加全局任务监听器,这个问题就迎刃而解了,按照上面的思路,自定义一个类如代码清单 11-29 所示。

代码清单 11-29 ShareniuUserTaskParserHandler.java

```

1 public class ShareniuUserTaskParserHandler extends UserTaskParseHandler{
2     protected void executeParse(BpmnParse bpmnParse, UserTask userTask) {
3         super.executeParse(bpmnParse, userTask);
4         TaskDefinition taskDefinition = (TaskDefinition)
5             bpmnParse.getCurrentActivity().getProperty("taskDefinition");
6         ActivitiListener activitiListener = new ActivitiListener();
7         activitiListener.setEvent("create");
8         activitiListener.setImplementationType("delegateExpression");
9         activitiListener.setImplementation("${shareniuTaskListener}");
10        taskDefinition.addTaskListener("create", bpmnParse.getListenerFactory()
11            .createDelegateExpressionTaskListener(activitiListener));
12    }
13 }

```

executeParse 方法的处理步骤如下。

(1) 第 3 行调用父类的 executeParse 方法,完成 userTask 对象的解析工作。

(2) 第 4~5 行获取 taskDefinition 对象。

(3) 第 6 行实例化 ActivitiListener 类。

(4) 第 7 行设置任务监听器的事件类型为 create(根据自己的业务需求进行设置,也可以是 complete 等)。

(5) 第 8 行指定该任务监听器的创建方式为 delegateExpression,因为在实际项目开发中,通常会将项目中的类交给 Spring 进行管理。

(6) 第 9 行设置了任务监听器的表达式。

(7) 第 10~11 行将自定义的任务监听器设置到 taskDefinition 对象中。

接下来定义一个任务监听器,如代码清单 11-30 所示。

代码清单 11-30 ShareniuTaskListener.java

```

1 @Service("shareniuTaskListener") //将该类交给 Spring 管理
2 public class ShareniuTaskListener implements TaskListener {
3     public void notify(DelegateTask delegateTask) {
4         System.out.println("shareniu"); //输出
5     }
6 }

```

然后将 ShareniuUserTaskParserHandler 类注入流程引擎配置类,如代码清单 11-31 所示。

代码清单 11-31 activiti.cfg.xml

```

1 <bean id="processEngineConfiguration"
2     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <property name="customDefaultBpmnParseHandlers"><!-- 通过该属性进行自定义对象解析器注入工作 -->
4     <list>

```

```

5      <bean class = "com.shareniu.chapter11.ShareniuUserTaskParserHandler" />
6    </list>
7  </property>
8 </bean>

```

对于 ShareniuTaskListener 类来说,该类可以交给 Spring 进行管理,也可以在流程实例运行时通过变量方式设置,如代码清单 11-32 所示。

代码清单 11-32 App.java

```

1 Map<String, Object> map = new HashMap<String, Object>();
2 map.put("shareniuTaskListener", new ShareniuTaskListener());
3 runtimeService.startProcessInstanceById("extensionOperationProcess:1:15004",map);

```

所有的操作执行完毕就可以部署一个带有任务节点的流程文档,并启动该流程实例,这样流程实例运转到流程文档中任意一个任务节点时,如不出意外,控制台的打印信息为:shareniu。

11.10 Activiti 新特性之字段注射模式

根据第 11.6.1 节中 applyFieldDeclaration 方法的处理逻辑可知,如果 field 为空并且 throwExceptionOnMissingField 参数值为空 true 则程序直接报错,该参数值默认为 true。换言之,如果在流程文档中为监听器定义了“字段”信息,但是没有在监听器(类)中定义这些字段,则程序就会报错。很显然,Activiti 在这里的设计有点不太人性化,Activiti 在 5.21 版本增加了“字段注射模式”,对应 DelegateExpressionFieldInjectionMode 枚举类,该类定义了如下三种模式。

(1) COMPATIBILITY: 通用类型(默认)。如果在流程文档中为监听器配置了“字段”信息,则监听器中必须定义这些字段,否则程序报错。

(2) MIXED: 混合类型。如果在流程文档中为监听器定义了“字段”信息,监听器中可以不定定义这些字段。

(3) DISABLED: 无效类型。预留字段,暂时没有提供实现。

以上三种模式可以通过流程引擎配置类的开关属性 delegateExpressionFieldInjectionMode 进行设置。

注意

字段注射模式目前仅应用于 serviceTask 的行为类 ServiceTaskDelegateExpression-ActivityBehavior 中。

第12章

Activiti 之设计模式

12.1 命令模式说明

之前章节中,反复提到 Activiti 将客户端所有的请求操作组装为一个一个的命令类,其内部采用命令模式执行这些命令类。对于命令类而言,Activiti 是如何设计的呢?开发人员如何优雅调度自定义命令类呢?Activiti 中的事务传播行为有哪些?又该如何合理的运用呢?经过本章的学习,对于上述问题会有一个新的认识。

12.1.1 命令模式的结构说明

接下来简单讲解命令模式,命令模式的结构图如图 12-1 所示。

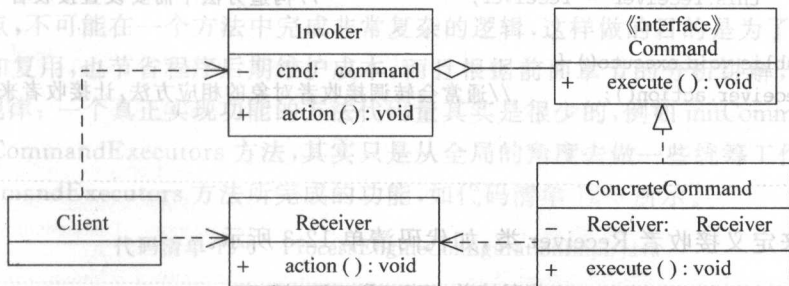


图 12-1 命令模式的结构图

- (1) Command: 命令定义接口,声明一系列的执行方法,该案例中定义了一个 execute 方法。
- (2) ConcreteCommand: 对 Command 接口中定义的方法进行实现,其内部持有接收者 Receiver 实例,并委托接收者完成命令要求实现的功能。如果开发人员打算简化 ConcreteCommand 类的实现,可以让 ConcreteCommand 类去除接收者 Receiver 类的引用,

直接完成命令要实现的功能。

(3) Receiver: 接收者, 真正执行命令的对象, 任何一个类或者接口都可能成为命令接收者, 标准的命令模式中, 只有该类才可以实现命令中的方法, 开发人员也可以将该类中的所有实现移植到 ConcreteCommand 类中, 从而将 Receiver 和 ConcreteCommand 类进行合并以简化烦琐的操作。

(4) Invoker: 命令调用者, 内部持有一系列的命令对象, 该类为客户端调用命令对象的入口。

(5) Client: 负责创建命令对象并为命令对象设置接收者, 然后调度 Invoker 类执行命令。

注意

这里所说的 Client 不是常规意义的客户端调用者对象, 而是对命令和接收者进行组装的对象, 为了更加容易理解, 可以将其称为命令装配者。

12.1.2 命令模式实战

(1) 首先定义一个命令接口, 如代码清单 12-1 所示。

代码清单 12-1 Command.java

```
9 public interface Command {    //命令接口定义,所有的命令对象均需要实现该接口
10     void execute();          // 执行方法
11 }
```

(2) 接下来定义命令接口的具体实现类, 如代码清单 12-2 所示。

代码清单 12-2 ConcreteCommand.java

```
1 public class ConcreteCommand implements Command{
2     private Receiver receiver = null;          //持有相应的接收者对象
3     public ConcreteCommand(Receiver receiver){
4         this.receiver = receiver;              //构造方法中需要设置接收者
5     }
6     public void execute() {
7         receiver.action();                      //通常会转调接收者对象的相应方法,让接收者来真正执行功能
8     }
9 }
```

(3) 再来定义接收者 Receiver 类, 如代码清单 12-3 所示。

代码清单 12-3 Receiver.java

```
1 public class Receiver {
2     public void action(){                      //真正执行命令的方法
3         System.out.println("分享牛执行操作");
4     }
5 }
```

(4) 定义命令调用者 Invoker 类,如代码清单 12-4 所示。

代码清单 12-4 Invoker.java

```
1 public class Invoker {  
2     private Command command = null;           //持有命令对象  
3     public Invoker(Command command){  
4         this.command = command;  
5     }  
6     public void action(){                      //行动方法  
7         command.execute();                    //命令调度者负责命令的调度工作  
8     }  
9 }
```

(5) 最后定义 Client 类,如代码清单 12-5 所示。

代码清单 12-5 Client.java

```
1 public class Client {  
2     public static void main(String[] args) {  
3         assemble();                          //客户端使用者调用,不是上面所说的 Client 对象  
4     }  
5     public static void assemble() {           //该方法就是上面所说的 Client 对象  
6         Receiver receiver = new Receiver();  //创建接收者  
7         Command command = new ConcreteCommand(receiver); //创建命令对象,设定它的接收者  
8         Invoker invoker = new Invoker(command); //创建请求者,把命令对象设置进去  
9         invoker.action();                     //执行方法  
10    }  
11 }
```

12.2 Activiti 命令模式

了解了命令模式的相关知识之后,接下来讲解 Activiti 是如何灵活运用命令模式的。首先明确一点,不可能在一个方法中完成非常复杂的逻辑,这样做的目的是为了使方法功能尽可能单一和复用,也节省程序后期维护成本,而且根据前面章节的分析讲解,或多或少也会发现一些规律:一个真正实现功能的方法代码量其实是很少的,例如 initCommandExecutor 方法,而 initCommandExecutors 方法,其实只是从全局的角度去做一些统筹工作。接下来,分析 initCommandExecutors 方法所完成的功能,如代码清单 12-6 所示。

代码清单 12-6 ProcessEngineConfigurationImpl.java

```
1 protected void initCommandExecutors() {  
2     initDefaultCommandConfig();  
3     initSchemaCommandConfig();  
4     initCommandInvoker();  
5     initCommandInterceptors();  
6     initCommandExecutor();  
7 }
```

将该方法的具体步骤总结如下。

- (1) 第 2 行初始化默认命令配置信息。
- (2) 第 3 行初始化 Schema 命令配置信息。
- (3) 第 4 行初始化命令调用者。
- (4) 第 5 行初始化命令拦截器。
- (5) 第 6 行初始化命令执行器。

12.2.1 初始化命令配置类

初始化默认命令配置信息和初始化 Schema 命令配置信息的具体实现如代码清单 12-7 所示。

代码清单 12-7 ProcessEngineConfigurationImpl.java 和 CommandConfig.java

```

1  ProcessEngineConfigurationImpl.java
2  protected void initDefaultCommandConfig() {
3      if (defaultCommandConfig == null) { //defaultCommandConfig 开关属性
4          defaultCommandConfig = new CommandConfig(); //实例化 CommandConfig 类
5      }
6  }
7  private void initSchemaCommandConfig() {
8      if (schemaCommandConfig == null) { //schemaCommandConfig 开关属性
9          //实例化 CommandConfig 的同时并调用 transactionNotSupported 方法
10         schemaCommandConfig = new CommandConfig().transactionNotSupported();
11     }
12 }
13 CommandConfig.java
14 public CommandConfig transactionNotSupported() {
15     CommandConfig config = new CommandConfig();
16     config.contextReusePossible = false;
17     config.propagation = TransactionPropagation.NOT_SUPPORTED;
18     return config;
19 }

```

在上述代码中,第 2 行定义的 initDefaultCommandConfig 方法首先判断 defaultCommandConfig 开关属性是否为空,如果为空,则第 4 行实例化 CommandConfig 类。

第 7 行定义的 initSchemaCommandConfig 方法首先判断 schemaCommandConfig 开关属性是否为空,如果为空,第 10 行实例化 CommandConfig 类并调用 transactionNotSupported 方法。

第 15 行实例化 CommandConfig 类,第 16~17 行设置 config 对象的属性值。CommandConfig 类内部持有事务的传播行为定义类 TransactionPropagation,以及命令上下文类是否可以继续使用的标识变量 contextReusePossible,该类的类图如图 12-2 所示。

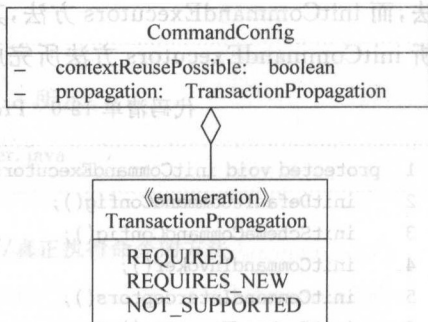


图 12-2 CommandConfig 类图

12.2.2 Activiti 事务传播行为

Activiti 事务传播行为在本书前面章节中还没有具体分析过, TransactionPropagation 类中定义的事务传播行为可以分为三种, 三种事务传播行为的讲解均是基于 Activiti 与 Spring 框架整合的基础之上。

(1) REQUIRED, 支持当前的事务, 如果当前方法中没有使用事务, 则新建一个事务去执行, 这是最常见的选择。使用下面的例子对事务传播行为进行通俗易懂的讲解, 如代码清单 12-8 所示。

代码清单 12-8 事务传播行为之 REQUIRED

```

1 public class ShareniuA{
2     public void actionB() {
3         shareniuB.action();
4     }
5 }
6 public class ShareniuB{
7     public void action() {
8         throw new RuntimeException("ShareniuB throw exception"); //抛出异常信息
9     }
10 }
11 public class ShareniuC{
12     public void actionB() {
13         try {
14             shareniuB.action();
15         } catch (RuntimeException e) {
16             }}}

```

对于上述情况, 第 2 行 ShareniuA 类中的 actionB 方法的事务传播行为将其定义为 REQUIRED, 第 7 行 ShareniuB 类中的 action 方法的事务传播行为将其定义为 REQUIRED, 这时直接调用 ShareniuA 类中的 actionB 方法就会捕获到 ShareniuB 类中 action 方法抛出的运行时异常, 则事务就会正常进行回滚, 道理很简单, 因为这两个方法均处于同一个事务中。如果 ShareniuB 类中 action 方法事务传播行为依然是 REQUIRED, 第 11~16 行 ShareniuC 类的 actionB 方法事务传播行为是 REQUIRED, 并在该方法中捕获异常, 这时执行 ShareniuC 类中的 actionB 方法, 程序就会报错, 如代码清单 12-9 所示。

代码清单 12-9 抛出异常信息

```

org.springframework.transaction.UnexpectedRollbackException: Transaction rolled back because
it has been marked as rollback-only.

```

出现这种异常信息的原因是什么呢? 不妨分析一下, 当 ShareniuB 类中的 action 方法抛出异常之后, 那么 ShareniuB 类会把当前的事务状态标记为回滚, 但是在 ShareniuC 类中捕获了这个异常并对其进行处理, 认为当前的事务是可以正常提交的, 由于这两个方法使用了同一个事务, 此时此刻就出现了前后不一致的情况, 正因为如此, 抛出了上述代码中的异

常信息。

(2) **REQUIRES_NEW**, 新建事务, 如果当前方法存在事务, 就把当前事务挂起。同样以上述代码为例进行说明, 继续使用 **ShareniuC** 类 (该类的事务传播行为为 **REQUIRED**), 修改 **ShareniuB** 类中的 **action** 方法事务传播行为为 **REQUIRES_NEW**, 这个时候再次执行 **ShareniuC** 类中的 **actionB** 方法, 程序是可以正常运行的, 为什么呢? 因为 **ShareniuC** 类调用 **ShareniuB** 类中的方法时, **ShareniuB** 类中的 **action** 方法会重新创建一个新的事务执行, 因此当 **action** 方法抛出异常之后, 仅仅是把新创建的事务进行回滚, 并不会影响到 **ShareniuC** 类中的事务, 所以 **ShareniuC** 类是可以正常的进行事务的提交或者回滚, 因为该方式操作的是两个独立的事务, 两个事务之间不会相互影响。

(3) **NOT_SUPPORTED**, 以非事务方式执行操作, 如果当前存在事务, 就把当前事务挂起。该方式可以参考 **REQUIRES_NEW** 的讲解。

12.2.3 Spring 事务拦截器

实际项目开发中经常使用 Spring 框架对事务进行管理并对事务的传播行为进行配置, 所以接下来详细分析 Spring 框架是如何接管 Activiti 中的事务传播行为, 进而从侧面验证上述所说的结论是否正确, 其相关实现如代码清单 12-10 所示。

代码清单 12-10 SpringTransactionInterceptor.java

```

1  protected PlatformTransactionManager transactionManager;
2  public <T> T execute(final CommandConfig config, final Command<T> command) {
3      TransactionTemplate transactionTemplate = new TransactionTemplate(transactionManager);
4      transactionTemplate.setPropagationBehavior(getPropagation(config));
5      T result = transactionTemplate.execute(new TransactionCallback<T>() {
6          public T doInTransaction(TransactionStatus status) {
7              return next.execute(config, command);
8          }
9      });
10     return result;
11 }
12 private int getPropagation(CommandConfig config) {
13     switch (config.getTransactionPropagation()) {
14         case NOT_SUPPORTED:
15             return TransactionTemplate.PROPROPAGATION_NOT_SUPPORTED;
16         case REQUIRED:
17             return TransactionTemplate.PROPROPAGATION_REQUIRED;
18         case REQUIRES_NEW:
19             return TransactionTemplate.PROPROPAGATION_REQUIRES_NEW;
20         default:
21             throw new ActivitiIllegalArgumentException("Unsupported transaction ");
22     }
23 }

```

SpringTransactionInterceptor 类设计的非常精巧, 通过类名就可以看出该类是一个 Spring 事务拦截器, 上述代码中, 通过第 12~21 行的处理逻辑可知 Spring 框架会获取

CommandConfig 类中定义的所有事务传播行为,然后将其转换为 Spring 框架中支持的事务传播行为。以上代码还附带透露了一个信息,目前 Activiti 只支持以上三种事务传播行为,其他事务传播行为流程引擎,暂不支持。

在使用 Spring 时不同平台的事务管理器都需要实现 PlatformTransactionManager 接口如第 1 行的定义所示,如果期望使用 JDBC 方式来处理事务,则事务管理器对应 DataSourceTransactionManager 类,构造事务管理器时必须传入一个 DataSource 实例对象,这样平台就知道如何与数据库进行交互,为了使平台的事务管理器对客户端来说是透明的,可以直接使用 TransactionTemplate,使用 TransactionTemplate 需要传入一个 PlatformTransactionManager 实例对象,这样程序就可以很方便地构造 TransactionTemplate 实例对象,而不用关心平台到底使用了什么事务管理器。

execute 方法首先在第 3 行实例化事务模板类 TransactionTemplate,然后第 4 行调用 getPropagation 方法将 Activiti 中的事务传播行为转化为 Spring 中的事务传播行为,并将转化后的事务传播行为类型设置到 transactionTemplate 对象中。第 7 行开始执行当前命令拦截器中的下一个命令拦截器,关于 SpringTransactionInterceptor 类添加到命令拦截器的过程稍后进行讲解。

12.2.4 初始化命令调度者

initCommandInvoker 方法用于初始化命令调度者,该方法的定义如代码清单 12-11 所示。

代码清单 12-11 ProcessEngineConfigurationImpl.java

```
1 protected void initCommandInvoker() {  
2     if (commandInvoker == null) {  
3         commandInvoker = new CommandInvoker();  
4     }  
5 }
```

如果 commandInvoker 开关属性为空,则直接实例化 CommandInvoker 类,该类非常重要,直接调度命令类。

建议

可以通过 commandInvoker 开关属性设置自定义命令调度者,进而对执行的命令进行控制。

12.2.5 初始化命令上下文工厂

ProcessEngineConfigurationImpl 类中的 initCommandContextFactory 方法用于初始化命令上下文工厂,该方法的相关实现如代码清单 12-12 所示。

代码清单 12-12 ProcessEngineConfigurationImpl.java

```
1 protected void initCommandContextFactory() {  
2     if (commandContextFactory == null) {
```

```

3   commandContextFactory = new CommandContextFactory();
4   }
5   commandContextFactory.setProcessEngineConfiguration(this);
6   }

```

在上述代码中,第2行判断 commandContextFactory 开关属性是否为空,如果该属性为空,第3行实例化 CommandContextFactory 类,第5行将 ProcessEngineConfigurationImpl 实例对象设置到 commandContextFactory 中。

CommandContextFactory 类用于创建 CommandContext 实例对象。

注意

第5行的 this 为 ProcessEngineConfigurationImpl 实例对象。

12.2.6 初始化命令拦截器

initCommandInterceptors 方法主要用于初始化命令拦截器链,该方法的具体实现如代码清单 12-13 所示。

代码清单 12-13 ProcessEngineConfigurationImpl.java

```

1   protected void initCommandInterceptors() {
2       if (commandInterceptors == null) {
3           commandInterceptors = new ArrayList<CommandInterceptor>();
4           if (customPreCommandInterceptors != null) { // 开关属性
5               commandInterceptors.addAll(customPreCommandInterceptors);
6           }
7           commandInterceptors.addAll(getDefaultCommandInterceptors());
8           if (customPostCommandInterceptors != null) {
9               commandInterceptors.addAll(customPostCommandInterceptors);
10          }
11          commandInterceptors.add(commandInvoker);
12      }
13  }

14  protected Collection<> getDefaultCommandInterceptors() {
15      List<CommandInterceptor> interceptors = new ArrayList<CommandInterceptor>();
16      interceptors.add(new LogInterceptor());
17      CommandInterceptor transactionInterceptor = createTransactionInterceptor();
18      if (transactionInterceptor != null) {
19          interceptors.add(transactionInterceptor);
20      }
21      interceptors.add(new CommandContextInterceptor(commandContextFactory, this));
22      return interceptors;
23  }

24  protected abstract CommandInterceptor createTransactionInterceptor();

```

将该方法的具体实现步骤以及功能总结如下。

(1) 第2行判断 commandInterceptors 开关属性是否为空,如果该属性值不为空,则使

用客户端自定义的命令拦截器集合,否则初始化默认命令拦截器集合,客户端可以设置该属性值,这依然是 Activiti 的一贯风格,预留回路,灵活扩展。如果客户端觉得默认命令拦截器不符合业务需求,可通过设置该开关属性将自定义的命令拦截器集合注入流程引擎配置类。

(2) 第 4~6 行获取并添加前置命令拦截器集合。

(3) 第 7 行调用 `getDefaultCommandInterceptors` 方法获取并添加内置命令拦截器集合。

`getDefaultCommandInterceptors` 方法的处理逻辑为:首先在第 16 行添加日志拦截器,然后第 17 行调用 `createTransactionInterceptor` 方法获取事务拦截器(上文所述的 Spring 事务拦截器就是在这里进行获取并添加到该集合中的),`createTransactionInterceptor` 方法是抽象方法,该方法需要交给具体的子类进行实现,Activiti 与 Spring 框架整合时,客户端可以直接构造 `SpringProcessEngineConfiguration` 类进行使用,该类继承 `ProcessEngineConfigurationImpl` 类,因此必须对父类中的抽象方法 `createTransactionInterceptor` 进行实现。如果开发人员没有接触过 Activiti 源码,可能还体会不到 Activiti 框架的良苦用心,为了功能更强大,配置更灵活,Activiti 做了很多工作,这一点值得学习和借鉴,Spring 引擎配置类 `SpringProcessEngineConfiguration` 的相关实现如代码清单 12-14 所示。

代码清单 12-14 SpringProcessEngineConfiguration.java

```
1 protected CommandInterceptor createTransactionInterceptor() {
2     if (transactionManager == null) {
3         throw new ActivitiException("transactionManager is required" + " otherwise");
4     }
5     return new SpringTransactionInterceptor(transactionManager);
6 }
```

在上述代码中,第 2 行对 `transactionManager` 开关属性进行非空校验,如果 `transactionManager` 为空则程序直接报错,试想一下 `transactionManager` 不存在如何进行事务的管理工作呢?最后第 5 行实例化 `SpringTransactionInterceptor` 类并返回,由此可知,事务拦截器的初始化工作非常重要,所幸 Activiti 框架对事务拦截器的添加预留回路,Spring 框架才可以灵活添加事务拦截器,如果 Activiti 框架没有预留添加事务拦截器的地方,就只能由客户端进行事务拦截器的添加以及注入工作,这将是一件非常痛苦的事情。

(4) 第 18~20 行获取 `transactionInterceptor` 并将其添加到集合。

(5) 第 21 行初始化命令上下文拦截器 `CommandContextInterceptor` 并将其添加到集合,命令上下文拦截器的初始化也是非常重要的一个环节。

(6) 第 8~10 行获取并添加后置命令拦截器集合。

(7) 第 11 行添加命令调用拦截器,命令调用拦截器 `CommandInvoker` 永远为拦截器链中最后一个执行节点,该类负责调度执行具体的命令类。

12.3 Activiti 职责链模式

在讲解命令拦截器如何被构造成链之前,先简单了解设计模式中的职责链模式,虽然该模式看上去很简单,但其实现却比较复杂。首先要明白一个问题,什么是链,对于链可以这

样理解。

(1) 链是一系列相同类型的集合。

(2) 链中的任何节点可以灵活拆分和组装。

职责链模式的结构,如图 12-3 所示。

(1) Handler: 定义处理请求的接口,内部持有 Handler 类型的变量。该接口中通常需要设置下一个处理节点,以及提供获取该节点的方法。

(2) ConcreteHandler: 实现 Handler 接口中定义的方法,该类接收到请求之后,可以选择自己处理请求还是将请求转发到下一个节点进行处理。

对于职责链的相关定义有了一定的了解之后,接下来详细分析 Activiti 是如何使用职责链将一系列的命令拦截器串起来使用的,关于命令拦截器链的初始化过程如代码清单 12-15 所示。

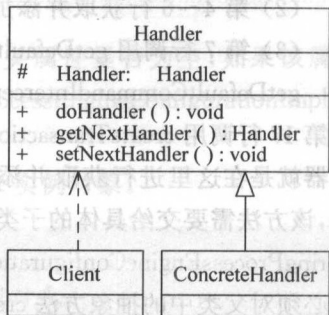


图 12-3 职责链的结构

代码清单 12-15 ProcessEngineConfigurationImpl.java

```

1  protected void initCommandExecutor() {
2      if (commandExecutor == null) {
3          CommandInterceptor first = initInterceptorChain(commandInterceptors);
4          commandExecutor = new CommandExecutorImpl(getDefaultCommandConfig(), first);
5      }
6  }
7  protected CommandInterceptor initInterceptorChain(List<CommandInterceptor> chain) {
8      if (chain == null || chain.isEmpty()) {
9          throw new ActivitiException("invalid command interceptor chain: " + chain);
10     }
11     for (int i = 0; i < chain.size() - 1; i++) {
12         chain.get(i).setNext(chain.get(i + 1));
13     }
14     return chain.get(0);
15 }

```

下面概括命令拦截器链的初始化步骤,并从中解释它所提供的功能。

(1) 第 2 行判断开关属性 commandExecutor。

如果 commandExecutor 开关属性值不为空则直接使用,否则开始构造命令拦截器链。同样这里还是给客户端预留了扩展的空间。

(2) 组装命令拦截器链。

第 3 行调用第 7 行定义的 initInterceptorChain 方法将一系列的命令拦截器组装成链,并返回链中的开始节点,该操作是为了方便后续程序自上而下执行命令拦截器。

initInterceptorChain 方法首先在第 8 行对命令拦截器集合 chain 进行非空校验,如果该集合为空或者集合中没有元素则程序直接报错,试想一下,如果命令拦截器集合为空则必

要的命令拦截器肯定缺失,不完整的命令拦截器链是无法使用的,最后第 11~13 行循环遍历 chain 集合并依此调用 CommandInterceptor 实例对象中的 setNext 方法构造命令拦截器链,由于传入的 chain 参数的数据结构为 List, List 集合是有序的,所以最终构造的命令拦截器链中的节点顺序和节点在 chain 集合中的顺序是一致的,命令拦截器链构造完毕之后第 14 行直接从 chain 集合中取出第一个元素作为该方法的返回值。

(3) 实例化命令执行器。

该过程比较简单,根据命令配置对象 defaultCommandConfig 以及命令拦截器链中的第一个节点实例化 CommandExecutorImpl 类,该类负责全局统筹命令拦截器的调用工作。

12.4 命令相关类职责

上面详细讲解了命令拦截器的初始化过程,下面再次总结该初始化过程涉及的类以及类的职责,如图 12-4 所示。

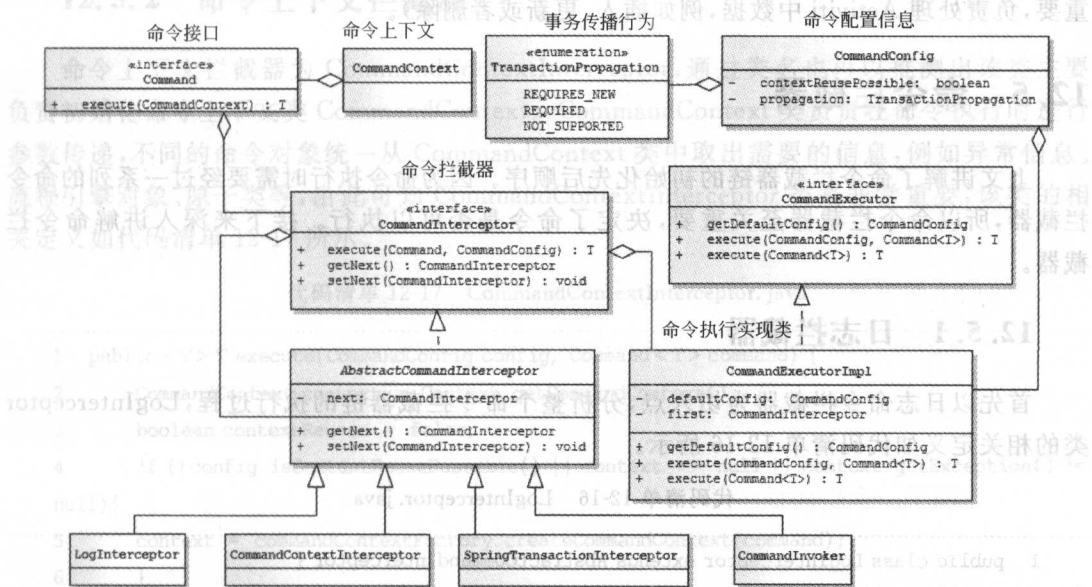


图 12-4 命令模式涉及的类图

- (1) Command: 命令接口,定义了 execute 方法,所有的命令类均需要实现该接口。
- (2) CommandContext: 命令上下文类,该类负责在命令中进行参数的传递,不同的命令对象统一从该类中取出需要的参数,并把执行结果通过该类传递给上层。
- (3) CommandConfig: 命令配置类,该类负责控制事务传播行为,检测命令上下文是否可用。
- (4) TransactionPropagation: 定义事务的传播行为。
- (5) CommandInterceptor: 命令拦截器,在命令类执行之前进行拦截,一个命令可以有多个拦截器,这一系列的拦截器最终构造为命令拦截器链,然后依此进行调用。
- (6) AbstractCommandInterceptor: 对 CommandInterceptor 接口中定义的方法进行了

实现,内部持有命令拦截器 `CommandInterceptor` 实例对象,可以把该类理解为一个模板类。

器端 (7) `CommandExecutor`: 该接口定义了执行命令拦截器以及获取命令配置信息的方法。

目之 (8) `CommandExecutorImpl`: 对 `CommandExecutor` 接口中定义的方法进行了实现,内部持有命令拦截器 `CommandInterceptor` 实例对象,并负责调度执行具体的命令拦截器实例对象。

的理 (9) `LogInterceptor`: 日志拦截器,负责记录命令的开始和结束。

用赋 (10) `SpringTransactionInterceptor`: Spring 事务拦截器,负责将 Activiti 的事务传播行为转化为 Spring 中的事务传播行为,这样 Spring 框架就可以使用事务管理器接管 Activiti 中的事务。一个节点进行处理。

(11) `CommandInvoker`: 命令调度者,当程序执行到该类时,开始调度命令类的执行工作。

或则 (12) `CommandContextInterceptor`: 该类负责命令上下文 `CommandContext` 以及上下文 `Context` 的初始化工作,并在所有命令对象执行完毕,调用上下文的关闭方法(该类非常重要,负责处理 Activiti 中数据,例如插入、更新或者删除)。

12.5 命令拦截器

上文讲解了命令拦截器链的初始化先后顺序。因为命令执行时需要经过一系列的命令拦截器,所以命令拦截器至关重要,决定了命令是否可以执行。接下来深入讲解命令拦截器。

12.5.1 日志拦截器

首先以日志命令拦截器为切入点,分析整个命令拦截器链的执行过程, `LogInterceptor` 类的相关定义如代码清单 12-16 所示。

代码清单 12-16 `LogInterceptor.java`

```

1 public class LogInterceptor extends AbstractCommandInterceptor {
2     public <T> T execute(CommandConfig config, Command<T> command) {
3         if (!log.isDebugEnabled()) {
4             return next.execute(config, command);
5         }
6         log.debug("---- starting {} -----", command.getClass().getSimpleName());
7         try {
8             return next.execute(config, command);
9         } finally {
10            log.debug("---- {} finished -----", command.getClass().getSimpleName());
11        }
12    }
13 }

```

如果开发人员没有配置前置拦截器则日志拦截器作为拦截器链中的第一个拦截器执行,该拦截器的执行过程非常简单,如果开发人员配置不需要输出 DEBUG 级别的日志,则直接执行第 4 行调用下一个命令拦截器。否则开始输出日志信息,即第 6 行在调用下一个拦截器之前先输出日志信息,然后第 8 行调用下一个拦截器,最后在第 10 行 finally 代码块中再次输出日志。

finally 关键字是对 Java 异常处理模型的最佳补充,不管程序有无异常发生,finally 中的代码总会执行,使用 finally 可以维护对象的内部状态,例如资源的清理以及回收。

所有的命令拦截器一般直接继承 AbstractCommandInterceptor 抽象类,由于该抽象类中的 execute 方法并没有进行 throws 抛出异常声明,所以该类的任何子类在实现该方法时,方法签名中不可以声明 throws 抛出异常,因此如果 execute 方法在执行过程中出现异常,则需要开发人员捕获异常信息并进行处理,由于日志拦截器仅仅是记录日志而已,所以该类出现异常的几率不大,几乎无可能,因此该类无须进行异常的捕获工作。接下来分析下一个拦截器,即命令上下文拦截器。

12.5.2 命令上下文拦截器

命令上下文拦截器为 CommandContextInterceptor,通过类名也可以推测出该类主要负责初始化命令上下文类 CommandContext。CommandContext 类负责在命令执行时进行参数传递,不同的命令对象统一从 CommandContext 类中取出需要的信息,例如异常信息、流程引擎对象、原子类等,由此可知 CommandContextInterceptor 类是非常重要,该类的相关定义如代码清单 12-17 所示。

代码清单 12-17 CommandContextInterceptor.java

```

1 public <T> T execute(CommandConfig config, Command<T> command) {
2     CommandContext context = Context.getCommandContext();
3     boolean contextReused = false;
4     if (!config.isContextReusePossible() || context == null || context.getException() !=
5         null){
6         context = commandContextFactory.createCommandContext(command);
7     }
8     else {
9         contextReused = true;
10    }
11    try {
12        Context.setCommandContext(context);
13        Context.setProcessEngineConfiguration(processEngineConfiguration);
14        return next.execute(config, command);
15    } catch (Exception e) {
16        context.exception(e);
17    } finally {
18        try {
19            if (!contextReused) {

```



```

19 context.close();
20 }
21 } finally {
22 Context.removeCommandContext();
23 Context.removeProcessEngineConfiguration();
24 Context.removeBpmnOverrideContext();
25 }
26 }
27 return null;
28 }

```

这段代码虽然不是很复杂,但是仅从表面上理解很难明白其设计意图,先总览该方法,其实现功能考虑了如下几个方面。

(1) 获取命令上下文实例对象。

上面代码中,第2行直接根据 Context 类获取命令上下文实例对象“context”,这里“context”变量的名称容易造成误解,因为“context”为 CommandContext 类型的变量,可能使用 CommandContext 变量表示会更加恰当。

(2) 判断命令上下文是否需要初始化。

首先第3行设置 contextReused 变量值为 false,如果步骤(1)没有获取到命令上下文实例对象,或者命令配置信息中明确指定已经存在的命令上下文类是不可重复使用的(之前已经获取的命令上下文实例对象),或者已经存在的命令上下文对象中记录有异常信息,则第5行需要重新实例化命令上下文类。如果经过层层判断发现不需要重新实例化命令上下文类则第8行直接设置 contextReused 变量值为 true。

(3) 填充上下文实例对象。

第11~12行将命令上下文“context”对象以及流程引擎 processEngineConfiguration 对象设置到 Context 类中,这样命令拦截器或者命令就可以通过 Context 类获取已经填充的属性值。

(4) 调用下一个拦截器。

第13行开始调用下一个拦截器。

(5) 异常处理。

如果以上执行过程中任何一个环节出错,则需要将异常信息捕获并设置到“context”对象中,通过该操作可知 CommandContext 实例对象中的 exception 属性负责记录异常信息。

(6) 第19行关闭命令上下文。

该过程首先会将会话缓存中的数据更新到数据库,可以参考第15章。

(7) 第22~24行在 finally 代码块中移除上下文 Context 类中的一系列集合。

12.5.3 上下文类

在命令上下文拦截器中涉及了上下文 Context 类的属性填充以及移除工作,Context 类的相关定义如代码清单 12-18 所示。

代码清单 12-18 Context.java

```

1  protected static ThreadLocal<Stack<CommandContext>> commandContextThreadLocal;
2  public static CommandContext getCommandContext() { //获取命令上下文实例对象
3      Stack<CommandContext> stack = getStack(commandContextThreadLocal);
4      if (stack.isEmpty()) { //如果栈为空则返回空
5          return null;
6      }
7      return stack.peek(); //获取栈顶的元素
8  }
9  protected static <T> Stack<T> getStack(ThreadLocal<Stack<T>> threadLocal) {
10     Stack<T> stack = threadLocal.get(); //首先从集合中获取
11     if (stack == null) { //如果当前线程不存在元素
12         stack = new Stack<T>(); //实例化栈并将其设置到 threadLocal 集合中
13         threadLocal.set(stack);
14     }
15     return stack;
16 }

```

根据第1行 commandContextThreadLocal 变量的声明,可以看出 Context 类的管控范围,首先提供了一系列不同类型的线程安全副本栈,为了避免线程冲突,每个命令都会在一个独立的命令上下文中执行,例如第2行定义的 getCommandContext 方法,该方法首先调用 getStack 方法从当前线程中获取命令上下文,然后再获取栈顶的元素值,看到这里可能会有一个疑问,ThreadLocal 为何被限制为 Stack 类型,而不是 Map 或者其他集合类型呢?其实在了解了命令的执行过程之后,就不会有这样的疑问了,上文讲解了程序首先创建命令上下文实例对象,然后设置下一个需要执行的命令拦截器,最后移除命令上下文实例对象,而创建和移除命令上下文实例对象的过程刚好与入栈和出栈的原理类似,所以这里使用栈管理最合适不过了,并且代码看起来非常简洁,如果使用其他的数据类型则势必会增加不必要的系统开销,并且操作起来有些怪异。

12.5.4 创建命令上下文实例对象

CommandContextFactory.createCommandContext(command) 方法负责实例化命令上下文 CommandContext 类,具体实现如代码清单 12-19 所示。

代码清单 12-19 CommandContextFactory.java

```

1  public CommandContext createCommandContext(Command<?> cmd) {
2      return new CommandContext(cmd, processEngineConfiguration);
3  }

```

CommandContextFactory 类的职责比较单一,负责维护流程引擎配置类 ProcessEngineConfigurationImpl 和创建 CommandContext 实例对象。在上文的讲解中多次提到命令上下文类 CommandContext,下面重点讲解该类,该类的功能结构图如图 12-5 所示,核心定义如代码清单 12-20 所示。

代码清单 12-20 CommandContext.java

```
1 public CommandContext(Command command, ProcessEngineConfigurationImpl pcf) {
2     this.command = command;
3     this.processEngineConfiguration = pcf;
4     this.failedJobCommandFactory = pcf.getFailedJobCommandFactory();
5     sessionFactories = pcf.getSessionFactories();
6     this.transactionContext =
7     pcf.getTransactionContextFactory().openTransactionContext(this);
8 }
```

CommandContext	
#	cmd: Command
#	transactionContext: TransactionContext
#	sessionFactories: Map<Class<?>, SessionFactory>
#	sessions: Map<Class<?>, Session>
#	exception: Throwable
#	nextOperations: LinkedList<AtomicOperation>
#	closeListeners: List<CommandContextCloseListener>
#	attributes: Map<String, Object>
+	performOperation(InterpretableExecution, AtomicOperation): void
+	close(): void
+	addCloseListener(CommandContextCloseListener): void
+	flushSessions(): void
+	exception(Throwable): void
+	getSession(Class): T

图 12-5 CommandContext 类功能结构图

CommandContext 构造方法根据传入的参数对 CommandContext 实例对象进行属性填充,command 参数为即将要执行的命令类实例对象,pcf 为流程引擎配置类实例对象,根据 pcf 参数可以获取 Session 会话工厂类和事务上下文等。

CommandContext 类中的 exception 属性封装了命令拦截器执行过程中可能出现的异常信息,如果程序出现异常,则该类的 close 方法不会执行 flushSessions 操作。CommandContext 类负责管理所有的实体管理类并提供获取管理器实例对象的方法。第 6 行的 transactionContext 事务上下文属性是为一系列的 session 服务的,TransactionContext 类控制事务的提交以及回滚操作,那么可以从更高的层次思考 CommandContext 类,既然该类中持有 TransactionContext 实例对象,那么 CommandContext 类中肯定有一些方法涉及了数据的入库操作,真的是这样吗?关于这一点可以参考第 15 章。

12.5.5 命令调度者拦截器

接下来分析命令是如何被 CommandInvoker 调用执行的,具体实现如代码清单 12-21 所示。

代码清单 12-21 CommandInvoker.java

```
1 public <T> T execute(CommandConfig config, Command<T> command) {
2     return command.execute(Context.getCommandContext());
3 }
```

```
4 public CommandInterceptor getNext() {  
5     return null;  
6 }  
7 public void setNext(CommandInterceptor next) {  
8     throw new UnsupportedOperationException("CommandInvoker must last the chain");  
9 }
```

CommandInvoker 类为拦截器链中的最后一个节点,该类中不能设置下一个处理类并且 getNext 方法返回值永远为空,该类中的 execute 方法首先获取 CommandContext 实例对象,然后调用 command 中的 execute 方法。看到这里可能会有疑问:在命令上下文拦截器 CommandContextInterceptor 中 execute 方法的 finally 代码块中不是已经移除命令上下文实例对象了吗,这里再次获取命令上下文实例对象应该无法获取吧?这就涉及 Java 语言中 try 以及 finally 代码块执行的先后顺序,如图 12-6 所示。

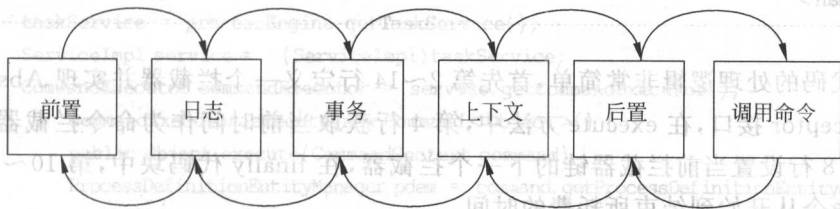


图 12-6 拦截器链执行先后顺序

当执行命令的时候,命令首先会被上面一系列的拦截器所拦截并按照拦截器链的顺序进行处理,经过前置拦截器(如果存在)、日志拦截器(开始记录日志)、事务拦截器(为执行命令的方法添加事务支持)、上下文拦截器(命令可以从命令上下文实例对象中获取必要的数数据)、后置拦截器(如果存在)、调用命令拦截器(命令开始被调用并执行),当命令执行完毕后,返回的数据从调用命令拦截器开始一层层向上传递并开始执行每一个拦截器中的 finally 代码块,该执行顺序与拦截器的执行顺序刚好相反。

12.6 自定义命令拦截器

在实际项目开发中,如果期望监控每一个命令类以及记录执行该命令类所耗费的时间,这时上文讲解的一系列拦截器无法满足需求,势必要自定义一个命令拦截器以满足需求,首先定义一个前置命令拦截器,如代码清单 12-22 所示。

代码清单 12-22 ShareniuInterceptor.java 和 activiti.cfg.xml

```
1 ShareniuInterceptor.java:
```

```
2 public class ShareniuInterceptor extends AbstractCommandInterceptor {  
3     public <T> T execute(CommandConfig config, Command<T> command) {  
4         long start = System.currentTimeMillis();    //开始时间  
5         try {  
6             System.out.println("需要执行的命令 " + command.getClass().getName());  
7             //打印需要执行的命令类  
8             //设置责任链中的下一请求处理者处理命令
```



```

8         return next.execute(config, command);
9     } finally {
10         long end = System.currentTimeMillis(); //获取当前时间为命令结束时间
11         System.out.println(end - start); //打印命令执行的总时长
12     }
13 }
14 }

15 activiti.cfg.xml:
16 <bean id="processEngineConfiguration"
17     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
18     <property name="customPreCommandInterceptors"><!-- 注入前置拦截器 -->
19         <list>
20             <bean class="com.shareniu.chapter12.ShareniuInterceptor"></bean>
21         </list>
22     </property>
23 </bean>

```

上面代码的处理逻辑非常简单,首先第 2~14 行定义一个拦截器并实现 `AbstractCommandInterceptor` 接口,在 `execute` 方法中,第 4 行获取当前时间作为命令拦截器的执行时间,然后第 8 行设置当前拦截器链的下一个拦截器,在 `finally` 代码块中,第 10~11 行直接计算当前命令从开始到结束所耗费的时间。

第 16~23 行将 `ShareniuInterceptor` 类作为前置拦截器并通过设置 `customPreCommandInterceptors` 开关属性将其注入到引擎配置类中。

扩展

`customPostCommandInterceptors` 开关属性可以实现后置拦截器的注入。

12.7 命令类调度入口

上面自定义了前置拦截器并将其注入流程引擎配置类,怎么验证其正确性呢?最好的办法就是执行一个命令来对其正确性进行校验,那么如何执行 Activiti 中的命令呢?在第 8.10.4 节中详细讲解了通过扩展流程引擎配置类来获取命令执行器 `CommandExecutor` 实例对象,但是这种获取方式不灵活,有没有一种更简单的获取方式呢?因为平时都是基于 Activiti 暴露的 API 对流程实例进行操作,所以接下来分析实例化服务类的同时是否为其填充 `commandExecutor` 属性值,如代码清单 12-24 所示。

代码清单 12-24 `ProcessEngineConfigurationImpl.java`

```

1 protected void initServices() {
2     initService(repositoryService);
3     initService(runtimeService);
4     ...//省略其他的服务类
5 }
6 protected void initService(Object service) {
7     if (service instanceof ServiceImpl) {

```

```
8      ((ServiceImpl)service).setCommandExecutor(commandExecutor);
9  }
10 }
```

在上述代码中,第2~4行分别调用 `initService` 方法进行处理,这一系列的服务类实例对象均是通过该方法将 `commandExecutor` 对象作为属性值注入进去的。

第7行 `initService` 方法首先判断 `service` 对象是否为 `ServiceImpl` 实例对象,如果是则执行第8行操作,了解了这点之后就可以很方便的通过上述任意一个服务类实例对象获取 `CommandExecutor` 实例对象。接下来验证上面的自定义拦截器是否生效,相关实现如代码清单 12-25 所示。

代码清单 12-25 App.java

```
1 public void addInputStreamTest() {
2     taskService = processEngine.getTaskService();
3     ServiceImpl service = (ServiceImpl)taskService;
4     CommandExecutor commandExecutor = service.getCommandExecutor();
5     commandExecutor.execute(new Command<Object>() {
6         public Object execute(CommandContext command) {
7             ProcessDefinitionEntityManager pdem = command.getProcessDefinitionEntityManager();
8             ProcessDefinitionEntity pde = pdem.findProcessDefinitionById("myss:12:140004");
9             System.out.println(pde);
10            return null;
11        }
12    });
13 }
```

在上述代码中,通过流程引擎实例对象获取任务服务对象 `taskService`(任意一个服务类实例对象即可),然后将其强制转化为 `ServiceImpl` 实例对象,转化的目的是为了更方便第4行通过 `ServiceImpl` 实例对象获取 `commandExecutor` 对象,第5~12行执行自定义命令类。

12.8 Activiti 事务

上文详细分析了 Spring 事务拦截器,那么 Activiti 是如何使用事务的呢?对于事务而言应该具备提交、回滚的功能。由于 Activiti 封装了 MyBatis 框架,所以接下来首先讲解 MyBatis 是如何进行事务管理的。

12.8.1 MyBatis 事务管理

MyBatis 框架将事务的操作抽象为 `Transaction` 接口,该接口的定义如图 12-7 所示。

简单分析上图中各个类的作用。

(1) `JdbcTransaction`,该类为 JDBC 事务管理机制,其内部使用 `java.sql.Connection` 实例对象完成事务的提交(`commit`)、回滚(`rollback`)操作。

(2) `ManagedTransaction`: 该类中的 `commit` 方法和 `rollback` 方法并没有任何实现。换

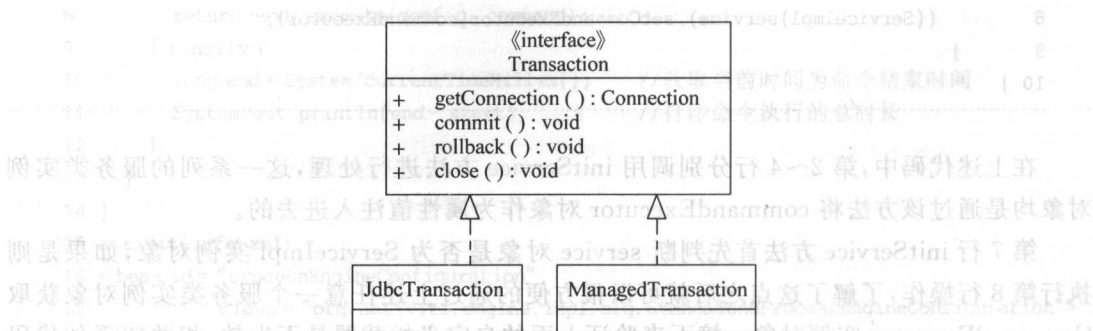


图 12-7 Transaction 接口

言之,这个类是不会控制事务的提交和回滚操作的,该类将事务的管理交给外部容器,例如 Spring 容器或者 JBOSS 容器。

12.8.2 事务上下文架构

接下来,分析 Activiti 是如何对事务进行操作的。TransactionContext 接口以及其子类如图 12-8 所示。

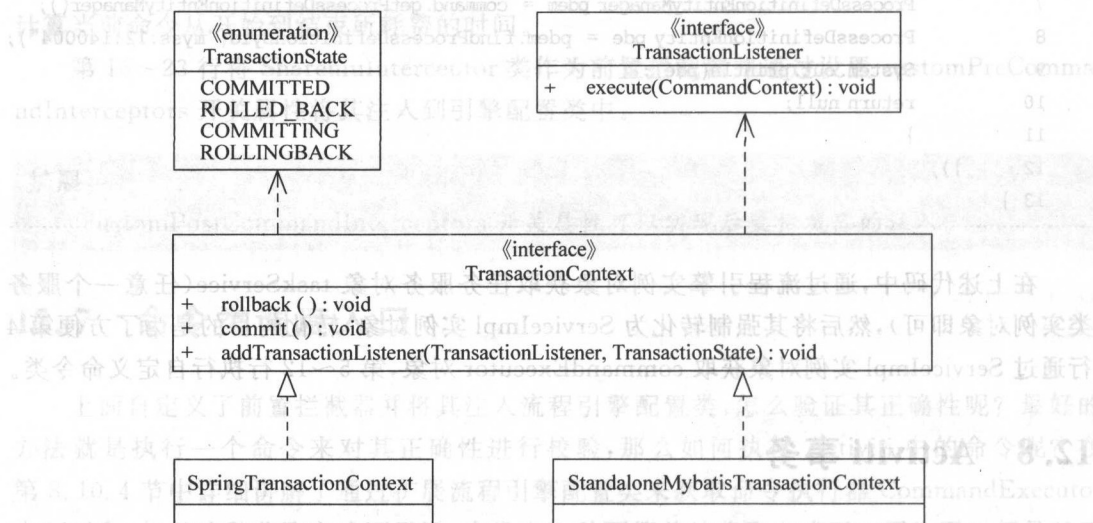


图 12-8 TransactionContext 接口

- (1) `TransactionState`: 该枚举类定义了事务可以执行的操作,例如提交 `COMMITTED`。
 - (2) `TransactionContext`: 该接口定义了事务的提交、回滚以及添加事务监听器的方法。
 - (3) `StandaloneMybatisTransactionContext`: 该类实现了 `TransactionContext` 接口,最终委托 `SqlSession` 实例对象完成事务的提交和回滚操作。
 - (4) `SpringTransactionContext`: 该类实现了 `TransactionContext` 接口,通过该类可以将事务交给 Spring 进行管理。
- 接下来,分析 Activiti 是如何初始化以上所列的类,如代码清单 12-26 所示。

代码清单 12-26 ProcessEngineConfigurationImpl.java

```
1 protected boolean transactionsExternallyManaged = false;
2 protected void initTransactionFactory() {
3     if (transactionFactory == null) {
4         if (transactionsExternallyManaged) {
5             transactionFactory = new ManagedTransactionFactory();
6         } else {
7             transactionFactory = new JdbcTransactionFactory();
8         }
9     }
10 }
```

在上述代码中,第 3 行判断 transactionFactory 开关属性是否为空,如果该属性值为空,则第 4 行判断 transactionsExternallyManaged 开关属性是否为 true,如果该属性值为 true,第 5 行实例化 ManagedTransactionFactory 类,否则第 7 行实例化 JdbcTransactionFactory 类。

根据上述的处理逻辑可知,如果使用 Spring 进行事务管理,只需要设置流程引擎配置类的 transactionsExternallyManaged 开关属性值为 true 即可,真的是这样吗?接下来分析 Spring 引擎配置类的相关实现,如代码清单 12-27 所示。

代码清单 12-27 SpringProcessEngineConfiguration.java

```
1 public SpringProcessEngineConfiguration() {
2     this.transactionsExternallyManaged = true;
3     ...//省略 Spring 自动部署流程资源策略类的初始化操作
4 }
```

的确如此,根据上述代码可知,第 2 行在构造方法中将 transactionsExternallyManaged 属性值设置为 true。

12.8.3 事务上下文工厂类

既然 TransactionContext 类负责操作事务,那么接下来分析该类是如何被实例化的? Activiti 将该类的实例化工作交给 TransactionContextFactory 类完成,TransactionContextFactory 类图如图 12-9 所示。

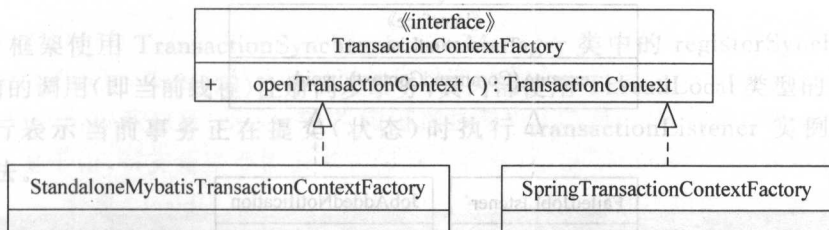


图 12-9 TransactionContextFactory 类图

首先分析 StandaloneMybatisTransactionContextFactory 类的初始化过程,如代码清单 12-28 所示。

代码清单 12-28 ProcessEngineConfigurationImpl.java

```

1 protected void initTransactionContextFactory() {
2     if (transactionContextFactory == null) {
3         transactionContextFactory = new StandaloneMybatisTransactionContextFactory();
4     }
5 }

```

在上述代码中,第 2 行判断 transactionContextFactory 开关属性是否为空,如果该属性值为空,则第 3 行直接实例化 StandaloneMybatisTransactionContextFactory 类。下面分析 SpringTransactionContextFactory 类的实例化过程,该类在 SpringProcessEngineConfiguration 类中进行实例化,如代码清单 12-29 所示。

代码清单 12-29 SpringProcessEngineConfiguration.java

```

1 protected void initTransactionContextFactory() {
2     if (transactionContextFactory == null && transactionManager != null) {
3         transactionContextFactory = new SpringTransactionContextFactory(transactionManager,
4             transactionSynchronizationAdapterOrder);
5     }
6 }

```

在上述代码中,第 2 行只有当前类的 transactionContextFactory 属性值为空且 transactionManager 属性值不为空时,则实例化 SpringTransactionContextFactory 类。

12.8.4 事务监听器

TransactionContext 类中定义了 addTransactionListener 方法,该方法主要用于控制事务状态发生变化时回调程序注册的事务监听器。首先分析事务监听器 TransactionListener,该类的类图如图 12-10 所示。

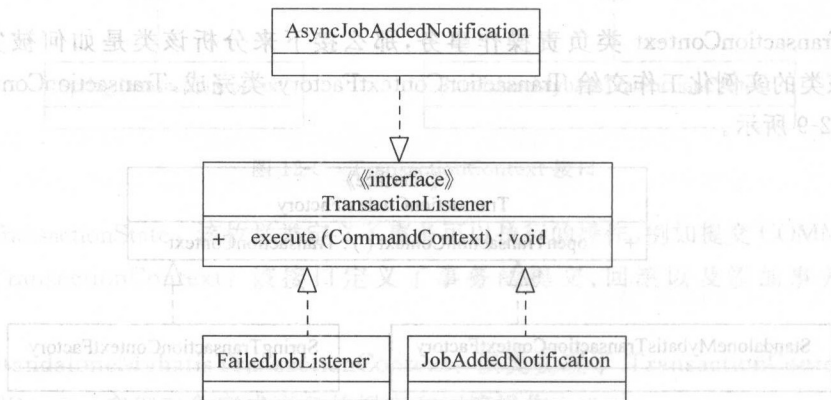


图 12-10 TransactionListener 类图

(1) JobAddedNotification: 添加作业。

(2) AsyncJobAddedNotification: 执行异步作业。

(3) FailedJobListener: 重新执行失败作业。

其中,FailedJobListener 类需要获取 FailedJobCommandFactory 实例对象,该实例对象的初始化如代码清单 12-30 所示。

代码清单 12-30 ProcessEngineConfigurationImpl.java

```
1 protected void initFailedJobCommandFactory() {  
2     if (failedJobCommandFactory == null) {  
3         failedJobCommandFactory = new DefaultFailedJobCommandFactory();  
4     }  
5 }
```

在上述代码中,第 2 行判断 failedJobCommandFactory 开关属性是否为空,如果该属性值不为空,则第 3 行直接实例化 DefaultFailedJobCommandFactory 类。

DefaultFailedJobCommandFactory 类用于获取 JobRetryCmd 命令类。

12.8.5 注册同步事务

本节重点讲解 SpringTransactionContext 类中的 addTransactionListener 方法的实现逻辑,该方法的相关实现如代码清单 12-31 所示。

代码清单 12-31 SpringTransactionContext.java

```
1 public void addTransactionListener(final TransactionState transactionState, final  
2 TransactionListener transactionListener) {  
3     if (transactionState.equals(TransactionState.COMMITTING)) {  
4         TransactionSynchronizationManager.registerSynchronization(new  
5 TransactionSynchronizationAdapter() {  
6             @Override  
7             public void beforeCommit(boolean readOnly) {  
8                 transactionListener.execute(commandContext);  
9             }  
10        });  
11    }  
12    ...//省略其他事务状态的注册  
13 }
```

Spring 框架使用 TransactionSynchronizationManager 类中的 registerSynchronization 方法为当前的调用(即当前线程)注册同步事务,其内部使用 ThreadLocal 类型的变量实现。第 3~11 行表示当前事务正在提交(状态)时执行 transactionListener 实例对象中的 execute 方法。

建议

可以结合上述的讲解自行学习 StandaloneMybatisTransactionContext 类。

流程虚拟机运转

本章节重点讲解流程虚拟机的运转过程，这可以说是 Activiti 的“高级”内容了。本章节中所有的知识点都围绕代码 `runtimeService.startProcessInstanceById(processDefinitionId, variables)` 展开。

13.1 流程实例运转入口

本案例以流程实例启动过程为例详细阐述其内部实现机制，相关实现如代码清单 13-1 所示。

代码清单 13-1 App.java

```
1 public void startProcessInstanceById() {
2     String processDefinitionId = "extensionOperationProcess:4:100004";
3     Map<String, Object> variables = new HashMap<String, Object>();
4     runtimeService.startProcessInstanceById(processDefinitionId, variables);
5 }
```

在学习流程实例启动之前首先思考一个问题，如果需要开发人员自己设计流程实例启动该如何实现？首先，需要定义一个流程实例启动接口和具体实现类，流程实例启动接口中需要定义一系列的启动方法，例如可以通过流程定义 key 启动、可以通过消息启动、也可以通过业务 id 或者租户 tenantId 启动，流程启动的方法还需要变量支持，由于变量的数量不确定，而且变量的类型也不固定，所以最好采用 `Map<String, Object>` 集合。

Activiti 中启动流程实例的一系列方法均定义在 `RuntimeService` 接口中。从上面的代码可以看出，流程实例启动操作步骤非常简单，首先第 2 行定位需要启动的流程定义 id 值（必须是已经部署并且没有挂起的流程才可以启动），如果流程实例启动时需要变量支持则可以使用第 3 行定义的 `variables` 集合进行存储，第 4 行通过调用 `RuntimeService` 接口中不

同的方法策略启动流程实例,该案例使用 `startProcessInstanceById` 方式启动流程实例,其他的启动方式可以结合实际应用场景自行选择。关于使用不同的方法策略启动流程实例的通用参数描述如下。

(1) `processDefinitionKey`: 流程定义 key,对应流程文档中 `process` 元素的 id 值。

(2) `businessKey`: 业务 Key,使用该参数后可以使流程实例与业务 `businessKey` 一一对应,该值通常用于绑定流程实例与项目中的业务,因此最好全局唯一(本书如果没有特别说明则业务键均指该值)。

(3) `tenantId`: 租户 id,可以用来标识哪些系统启动了流程实例,这样后续查询任务等操作时可以根据该值进行区分。租户 id 可以标识“A”系统,也可以标识“B”系统,具体使用方式可结合项目中的业务场景。

(4) `messageName`: 该参数值适用于消息方式启动流程实例,参数值对应流程文档中 `message` 元素的 `name` 值。

(5) `Map<String, Object>`: 该参数用于存储变量信息,虽然集合的 `value` 值是 `Object` 类型,但是并非支持 Java 语言中所有的 `Object` 类型,可以参考第 16.2 节。

13.2 启动流程实例命令类

了解了流程实例启动的方式以及通用参数描述之后,接下来首先找到启动流程实例的命令类 `StartProcessInstanceCmd`,该类的相关定义如代码清单 13-2 所示。

代码清单 13-2 StartProcessInstanceCmd.java

```
1 public ProcessInstance execute(CommandContext commandContext) {
2     DeploymentManager deploymentManager = commandContext.getProcessEngineConfiguration()
3         .getDeploymentManager();
4     ProcessDefinitionEntity processDefinition = null;
5     if (processDefinitionId != null) {
6         processDefinition =
7             deploymentManager.findDeployedProcessDefinitionById(processDefinitionId);
8         if (processDefinition == null) {
9             }
10        } else if (processDefinitionKey != null && (tenantId == null || "".equals(tenantId))) {
11            processDefinition =
12                deploymentManager.findDeployedLatestProcessDefinitionByKey(processDefinitionKey);
13            if (processDefinition == null) {
14                }
15        } else if (processDefinitionKey != null && tenantId != null && !"".equals(tenantId)) {
16            processDefinition = deploymentManager.findDeployedLatestProcessDefinitionByKeyAndTenantId(
17                processDefinitionKey, tenantId);
18            if (processDefinition == null) {
19                throw new ActivitiObjectNotFoundException("No process definition found for key ");
20            }
21        } else {
22            throw new ActivitiIllegalArgumentException("processDefinitionKey null");
23        }
```



```

24 if (deploymentManager.isProcessDefinitionSuspended(processDefinition.getId())) {
25     throw new ActivitiException("Cannot start process instance");
26 }
27 ExecutionEntity processInstance = processDefinition.createProcessInstance(businessKey);
28 initializeVariables(processInstance);
29 if (processInstanceName != null) {
30     processInstance.setName(processInstanceName);
31     commandContext.getHistoryManager().recordProcessInstanceNameChange(
32         processInstance.getId(), processInstanceName);
33 }
34 processInstance.start();
35 return processInstance;
36 }
37 protected void initializeVariables(ExecutionEntity processInstance) {
38     if (variables != null) {
39         processInstance.setVariables(variables);
40     }
41 }
42 public StartProcessInstanceCmd(String processDefinitionKey, String processDefinitionId,
43     String businessKey, Map<String, Object> variables) {
44     this.processDefinitionKey = processDefinitionKey; //流程定义 key
45     this.processDefinitionId = processDefinitionId; //流程定义 id
46     this.businessKey = businessKey;
47     this.variables = variables; //变量
48 }

```

流程实例启动的处理逻辑非常清晰,在深入讲解每一个实现细节之前,先总览整个方法如下。

(1) 第 2~3 行获取 DeploymentManager 实例对象,DeploymentManager 类负责管理所有的缓存处理类。

(2) 第 4~23 行获取 ProcessDefinitionEntity 实例对象,获取 ProcessDefinitionEntity 实例对象的意义何在呢? 第 10 章中讲解了所有节点和连线的定义信息分别对应流程虚拟机中的 ActivityImpl 实例对象和 TransitionImpl 实例对象,并最终存储在 ProcessDefinitionEntity 实例对象中,因为后续流程虚拟机运转时需要获取 ActivityImpl 实例对象和 TransitionImpl 实例对象的信息,所以 ProcessDefinitionEntity 实例对象的获取是非常重要的,该环节首先尝试从缓存中获取,如果缓存中不存在,则会再次执行流程文档的部署工作。

(3) 挂起流程验证,通过第 24 行的处理可以看出已经挂起的流程是不能执行启动操作的,如果强行启动已经挂起的流程,则第 25 行程序会报错。

(4) 第 27 行构造 ExecutionEntity 实例对象。该实例对象是流程实例运转的核心对象,负责流程实例的创建、启动、结束等操作,并持有 ProcessDefinitionEntity 实例对象,所以程序可以很方便通过 ExecutionEntity 实例对象获取 ProcessDefinitionEntity 实例对象,然后通过 ProcessDefinitionEntity 实例对象获取流程虚拟机中的 ActivityImpl 实例对象和 TransitionImpl 实例对象。

(5) 第 28 行调用 initializeVariables 方法设置流程实例变量,该变量存储在代码清单 13-1 中定义的 variables 集合。

(6) 第 29 行判断 processInstanceName 是否为空,如果不为空则第 30~32 行更新历史流程实例的名称。

(7) 第 34 行启动流程实例。

13.2.1 获取 ProcessDefinitionEntity 实例对象

接下来,深入分析 ProcessDefinitionEntity 实例对象的获取工作。上面代码中 ProcessDefinitionEntity 实例对象可以根据不同的方法策略进行获取,可以根据 processDefinitionId 值进行获取,可以根据 processDefinitionKey 值进行获取,也可以根据 processDefinitionKey 值和 tenantId 值联合起来进行获取,以上所说的三个值在 StartProcessInstanceCmd 类实例化的同时,已经通过构造方法初始化了,最终 Activiti 根据流程实例的启动策略调用 DeploymentManager 类中对应的方法获取 ProcessDefinitionEntity 实例对象,本案例以 findDeployedProcessDefinitionById 方法为例深入探究其内部实现机制,该方法的具体实现如代码清单 13-3 所示。

代码清单 13-3 DeploymentManager.java

```
1 public ProcessDefinitionEntity findDeployedProcessDefinitionById(String processDefinitionId) {
2     if (processDefinitionId == null) {
3         throw new ActivitiIllegalArgumentException("Invalid process definition id : null");
4     }
5     ProcessDefinitionEntity processDefinition = processDefinitionCache.get(processDefinitionId);
6     if (processDefinition == null) {
7         processDefinition = Context.getCommandContext().getProcessDefinitionEntityManager().
8             findProcessDefinitionById(processDefinitionId);
9         if (processDefinition == null) {
10             throw new ActivitiObjectNotFoundException("nodeployed" + "!", ProcessDefinition.class);
11         }
12         processDefinition = resolveProcessDefinition(processDefinition);
13     }
14     return processDefinition;
15 }
```

在上述代码中,第 2 行对 processDefinitionId 参数进行校验,如果该参数为空则程序报错;如果该参数不为空,则执行第 5 行代码尝试从缓存中获取 ProcessDefinitionEntity 实例对象,如果缓存中存在则该方法直接返回,如果缓存中不存在,则执行第 7~11 行委托 ProcessDefinitionEntityManager 实例对象获取 ProcessDefinitionEntity 实例对象,该实例对象最终通过调用 MyBatis 框架并根据 processDefinitionId 参数值作为查询条件从数据库 ACT_RE_PROCDEF 表中查询数据,如果从数据库中获取到数据则第 12 行委托 resolveProcessDefinition 方法进行流程文档的部署操作,如果数据库没有查询到数据则第 10 行程序直接报错,从这里的处理逻辑可以看出,缓存 ProcessDefinitionEntity 实例对象是非常有必要的,这也是性能优化的地方之一。

13.2.2 重新生成流程定义缓存数据

resolveProcessDefinition 方法主要用于部署流程文档,引擎为何委托该方法进行流程

文档的部署操作？试想一下通过 `ProcessDefinitionEntityManager` 实例对象从数据库中直接获取的 `ProcessDefinitionEntity` 实例对象的属性值仅仅是与 `ACT_RE_PROCDEF` 表中的字段值一一对应，而流程虚拟机最终需要通过 `ProcessDefinitionEntity` 实例对象获取 `ActivityImpl` 实例对象以及 `TransitionImpl` 实例对象，如果流程文档没有经过部署环节，则 `ActivityImpl` 实例对象和 `TransitionImpl` 实例对象是没有办法进行获取和赋值的，进而导致 `ProcessDefinitionEntity` 实例对象中部分属性值缺失，不完整的实例对象是无法供流程虚拟机使用的，该操作用来保证虚拟机中的各种实例对象可以正常初始化。了解了 `resolveProcessDefinition` 方法的设计意图和动机之后，接下来详细分析该方法的具体处理逻辑，如代码清单 13-4 所示。

代码清单 13-4 / `DeploymentManager.java`

```
1 public ProcessDefinitionEntity resolveProcessDefinition(ProcessDefinitionEntity processDefinition)
2 {
3     String processDefinitionId = processDefinition.getId();
4     String deploymentId = processDefinition.getDeploymentId();
5     processDefinition = processDefinitionCache.get(processDefinitionId);
6     if (processDefinition == null) {
7         DeploymentEntity deployment = Context.getCommandContext().getDeploymentEntityManager()
8             .findDeploymentById(deploymentId);
9         deployment.setNew(false);
10        deploy(deployment, null);
11        processDefinition = processDefinitionCache.get(processDefinitionId);
12        if (processDefinition == null) {
13            throw new ActivitiException("'" + processDefinitionId + "' in the cache");
14        }
15    }
16    return processDefinition;
17 }
```

该方法的处理逻辑非常清晰，第 2 行获取 `processDefinitionId` 值，第 4 行根据 `processDefinitionId` 值从缓存中获取对应的 `ProcessDefinitionEntity` 实例对象，如果没有从缓存中获取到数据，则开始执行第 6~14 行代码，其中第 6~7 行首先根据 `Context` 类获取 `CommandContext` 实例对象，然后通过该实例对象获取 `DeploymentEntityManager` 实例对象，最终通过 `DeploymentEntityManager` 实例对象获取 `DeploymentEntity` 实例对象，第 8 行设置 `DeploymentEntity` 实例对象的 `isNew` 属性值为“false”，在第 7.2.2 节详细讲解过设置该值为 false 的意义，第 9 行调用当前类中的 `deploy` 方法部署流程文档。

第 10 行操作是为了确保 `ProcessDefinitionEntity` 实例对象已经被成功添加到缓存中，因为流程文档部署涉及了元素解析以及对象解析，该过程是非常消耗性能，所以该操作非常有必要。

13.3 创建流程实例

至此为止 `ProcessDefinitionEntity` 实例对象的获取和对其进行属性填充的过程已经讲解完毕。代码清单 13-2 调用 `ProcessDefinitionEntity` 实例对象的 `createProcessInstance` 方

法创建 ExecutionEntity 实例对象,那么该实例对象是如何被创建的呢? 内部实现机制是什么? 这里以 processDefinition.createProcessInstance(businessKey)作为 ExecutionEntity 实例对象创建的切入点进行讲解,首先分析这个方法的时序图如图 13-1 所示。

约定

如果流程不存在分支或者多实例节点则 ExecutionEntity 实例对象与流程实例对象是等价的。

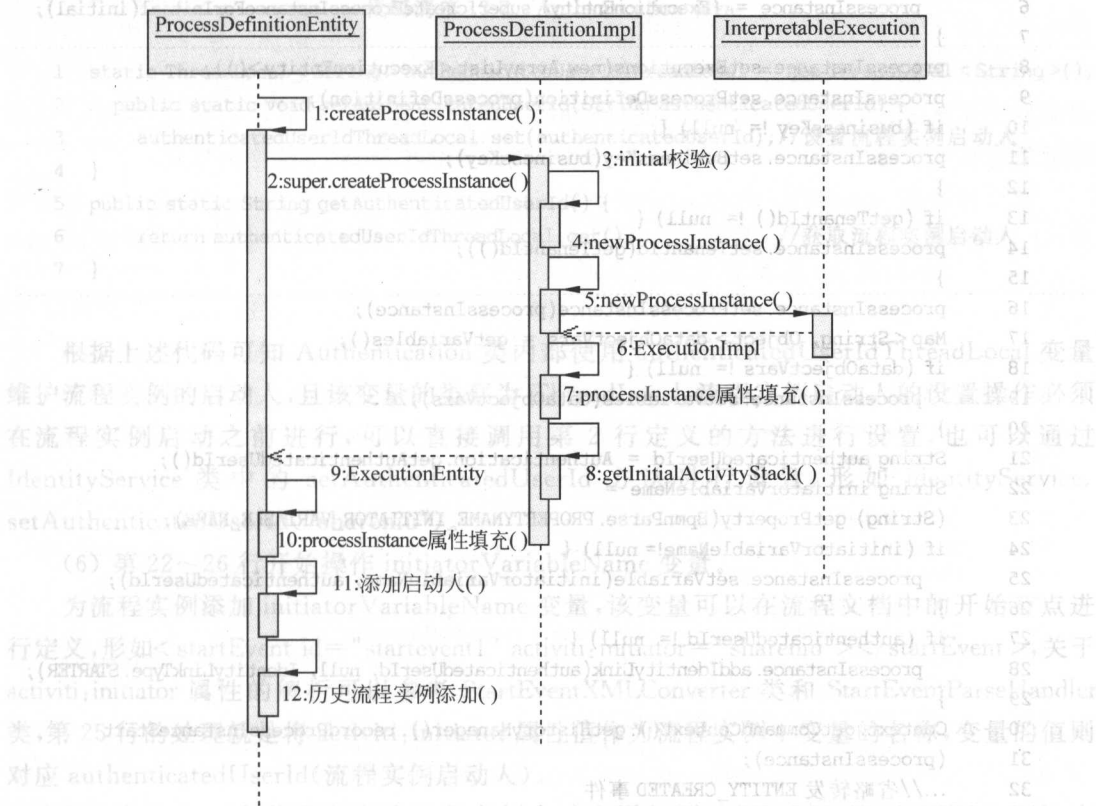


图 13-1 创建流程实例时序图

从 ProcessDefinitionEntity 类的 createProcessInstance 方法开始,通过该图可以一目了然地看到创建流程实例的全部处理过程,该处理过程比较复杂,需要结合源码逐个剖析解读,createProcessInstance 方法的相关实现如代码清单 13-5 所示。

代码清单 13-5 ProcessDefinitionEntity.java

```
1 public ExecutionEntity createProcessInstance(String businessKey) {
2     return createProcessInstance(businessKey, null);
3 }
```

根据上面的代码可以看出创建流程实例时,需要传入 businessKey 参数,上文讲解过该参数是业务标识位,最好全局唯一,以方便区分流程实例属于哪一个业务 key。该方法在第

2 行直接调用 `createProcessInstance(businessKey, null)` 方法创建 `ExecutionEntity` 实例对象,如代码清单 13-6 所示。

代码清单 13-6 ProcessDefinitionEntity.java

```

1 public ExecutionEntity createProcessInstance(String businessKey, ActivityImpl initial) {
2     ExecutionEntity processInstance = null;
3     if(initial == null) {
4         processInstance = (ExecutionEntity) super.createProcessInstance();
5     }else {
6         processInstance = (ExecutionEntity) super.createProcessInstanceForInitial(initial);
7     }
8     processInstance.setExecutions(new ArrayList<ExecutionEntity>());
9     processInstance.setProcessDefinition(processDefinition);
10    if (businessKey != null) {
11        processInstance.setBusinessKey(businessKey);
12    }
13    if (getTenantId() != null) {
14        processInstance.setTenantId(getTenantId());
15    }
16    processInstance.setProcessInstance(processInstance);
17    Map<String, Object> dataObjectVars = getVariables();
18    if (dataObjectVars != null) {
19        processInstance.setVariables(dataObjectVars);
20    }
21    String authenticatedUserId = Authentication.getAuthenticatedUserId();
22    String initiatorVariableName =
23        (String) getProperty(BpmnParse.PROPERTYNAME_INITIATOR_VARIABLE_NAME);
24    if (initiatorVariableName != null) {
25        processInstance.setVariable(initiatorVariableName, authenticatedUserId);
26    }
27    if (authenticatedUserId != null) {
28        processInstance.addIdentityLink(authenticatedUserId, null, IdentityLinkType.STARTER);
29    }
30    Context.getCommandContext().getHistoryManager().recordProcessInstanceStart
31        (processInstance);
32    ...//省略转发 ENTITY_CREATED 事件
33    }
34    return processInstance;
35 }

```

对上述代码所实现的功能进行如下总结。

(1) 第 2~7 行实例化 `ExecutionEntity` 类。该类的实例化逻辑非常简单,不管 `initial` 参数值是否为空,都会委托当前类 `ProcessDefinitionEntity` 的父类 `ProcessDefinitionImpl` 创建该实例对象,唯一的区别在于,如果 `initial` 参数不为空,则第 6 行使用 `ProcessDefinitionImpl` 类中 `createProcessInstanceForInitial(initial)` 方法;如果该参数为空,则第 4 行使用 `ProcessDefinitionImpl` 类中的 `createProcessInstance()` 方法。

(2) 第 8~16 行填充 `ExecutionEntity` 实例对象属性。`ExecutionEntity` 类为 `ACT_RU_EXECUTION` 表的映射实体类,所以 `ExecutionEntity` 类中定义的属性与 `ACT_RU_EXECUTION` 表中的字段一一对应。

(3) 获取 dataObject 元素信息。

第 17 行调用 getVariables 方法获取流程文档中定义的 dataObject 元素信息。dataObject 元素是 BPMN 的规范之一,Activiti 对其进行实现。

(4) 第 18~20 行将 dataObject 元素信息作为流程实例的变量进行处理。

(5) 第 21 行获取流程实例的启动人。

Authentication.getAuthenticatedUserId()负责获取流程实例的启动人,相关实现如代码清单 13-7 所示。

代码清单 13-7 Authentication.java

```
1 static ThreadLocal<String> authenticatedUserIdThreadLocal = new ThreadLocal<String>();
2 public static void setAuthenticatedUserId(String authenticatedUserId) {
3     authenticatedUserIdThreadLocal.set(authenticatedUserId); //设置流程实例启动人
4 }
5 public static String getAuthenticatedUserId() {
6     return authenticatedUserIdThreadLocal.get(); //获取流程实例启动人
7 }
```

根据上述代码可知 Authentication 类内部使用 authenticatedUserIdThreadLocal 变量维护流程实例的启动人,且该变量的类型为 ThreadLocal,流程实例启动人的设置操作必须在流程实例启动之前进行,可以直接调用第 2 行定义的方法进行设置,也可以通过 IdentityService 类中的 setAuthenticatedUserId 方法进行设置,形如 identityService.setAuthenticatedUserId("sharenui")。

(6) 第 22~26 行开始操作 initiatorVariableName 变量。

为流程实例添加 initiatorVariableName 变量,该变量可以在流程文档中的开始节点进行定义,形如< startEvent id="startevent1" activiti:initiator="sharenui"></startEvent>,关于 activiti:initiator 属性的解析可以参考 StartEventXMLConverter 类和 StartEventParseHandler 类,第 25 行的处理就是将 activiti:initiator 属性值作为流程实例中变量的名称,变量的值则对应 authenticatedUserId(流程实例启动人)。

(7) 第 27~29 行代码负责将流程实例启动人添加到 ACT_RU_IDENTITYLINK 表中。通过该操作可以发现流程实例启动操作只支持具体的人,不支持组方式,该过程的详细实现可以查看 ExecutionEntity 类中的 addIdentityLink 方法。

(8) 第 30~31 行将 ExecutionEntity 实例对象中的已知属性值添加会话缓存中,引擎最终将会话缓存中的数据刷新到历史流程实例表(ACT_HI_PROCINST)中。

扩展

ProcessDefinitionImpl 类中的 initial 属性初始化工作可以参考 StartEventParseHandler 类。

13.3.1 创建 ExecutionEntity 实例对象

ExecutionEntity 实例对象的创建工作委托给 ProcessDefinitionEntity 的父类 ProcessDefinitionImpl,相关实现如代码清单 13-8 所示。

代码清单 13-8 ProcessDefinitionImpl.java

```

1 public PvmProcessInstance createProcessInstance() {
2     if(initial == null) {
3         throw new ActivitiException("Process '" + name + "' has no default start activity");
4     }
5     return createProcessInstanceForInitial(initial);
6 }

```

在上述代码中,第 2 行对 ProcessDefinitionImpl 实例对象中的 initial 属性值进行非空校验,initial 属性为 ActivityImpl 类型,具体存储哪一个节点的信息呢?思考流程实例启动时最先执行流程文档中的哪一个节点呢?当然是开始节点,流程文档如果没有定义开始节点就好比无水之源,无本之木,所以这里的 initial 参数值就是开始节点的信息,关于该属性的赋值操作可以参考 StartEventParseHandler 类的 selectInitial 方法中的实现。

第 5 行委托 createProcessInstanceForInitial(initial) 方法创建 ExecutionEntity 实例对象(ExecutionEntity 类实现了 PvmProcessInstance 接口),该方法的定义如代码清单 13-9 所示。

代码清单 13-9 ProcessDefinitionImpl.java

```

1 public PvmProcessInstance createProcessInstanceForInitial(ActivityImpl initial) {
2     if(initial == null) {
3         throw new ActivitiException("Cannot start process instance");
4     }
5     InterpretableExecution processInstance = newProcessInstance(initial);
6     processInstance.setProcessDefinition(this);
7     processInstance.setProcessInstance(processInstance); //设置流程实例 id
8     processInstance.initialize();
9     InterpretableExecution scopeInstance = processInstance;
10    List<ActivityImpl> initialActivityStack = getInitialActivityStack(initial);
11    for (ActivityImpl initialActivity: initialActivityStack) {
12        if (initialActivity.isScope()) {
13            scopeInstance = (InterpretableExecution) scopeInstance.createExecution();
14            scopeInstance.setActivity(initialActivity);
15            if (initialActivity.isScope()) {
16                scopeInstance.initialize();
17            }
18        }
19    }
20    scopeInstance.setActivity(initial);
21    return processInstance;
22 }

```

下面概括该方法的处理逻辑。

(1) 第 2 行校验 initial 参数值。因为该方法可能会被 StartProcessInstanceByMessageCmd (消息开始节点)等其他类方法直接调用,所以这里进行参数值的校验工作是非常有必要的。

(2) 第 5 行实例化 InterpretableExecution 类。

该类的实例化工作委托 newProcessInstance(initial) 方法进行处理,该方法的具体实现如代码清单 13-10 所示。

代码清单 13-10 ProcessDefinitionEntity.java

```
1 protected InterpretableExecution newProcessInstance(ActivityImpl activityImpl) {  
2     ExecutionEntity processInstance = new ExecutionEntity(activityImpl);  
3     processInstance.insert();  
4     return processInstance;  
5 }
```

上面提到了 ProcessDefinitionEntity 类委托其父类 ProcessDefinitionImpl 创建 ExecutionEntity 实例对象, 尽管 newProcessInstance 方法的调用位于 ProcessDefinitionImpl 类中, 但也应该牢记程序当前的 this 指针指向 ProcessDefinitionEntity 类的具体实例对象, 而并非 ProcessDefinitionImpl 实例对象, 由于 ProcessDefinitionEntity 类重写了其父类中的 newProcessInstance 方法, 所以程序最终会调用 ProcessDefinitionEntity 类的 newProcessInstance 方法。

正如上面的代码所示, 该方法第 2 行实例化 ExecutionEntity 类, 第 3 行调用 processInstance 对象中的 insert 方法, 为该对象插入数据库做准备, 看到这里可能会有疑问, 实例化 ExecutionEntity 类时并没有进行属性的填充操作, 怎么能首先进行 insert 操作呢? 其实此处 insert 操作目的只是通知引擎 ExecutionEntity 类的实例化操作已经执行完毕, 并且已经做好了插入会话缓存的准备, 该过程可以参考第 15 章。

ExecutionEntity 类中构造方法的相关实现如代码清单 13-11 所示。

代码清单 13-11 ExecutionEntity.java

```
1 protected StartingExecution startingExecution;  
2 public ExecutionEntity(ActivityImpl activityImpl) {  
3     this.startingExecution = new StartingExecution(activityImpl);  
4 }
```

在上述代码中, 第 3 行根据 activityImpl 参数直接实例化 StartingExecution 类。

(3) 第 6 行 processInstance.setProcessDefinition(this) 操作主要为 ExecutionEntity 实例对象填充属性, 这里所说填充的属性有 processDefinition、processDefinitionId、processDefinitionKey, 这三个属性值的填充操作可以参考 ExecutionEntity 类中的 setProcessDefinition 方法, 如代码清单 13-12 所示。

代码清单 13-12 ExecutionEntity.java

```
1 public void setProcessDefinition(ProcessDefinitionImpl processDefinition) {  
2     this.processDefinition = processDefinition;  
3     this.processDefinitionId = processDefinition.getId();  
4     this.processDefinitionKey = processDefinition.getKey();  
5 }
```

(4) 第 8 行 processInstance.initialize() 方法主要是为了确保当前执行实例是否存在流程实例, 如果存在, 则需要初始化流程实例、执行实例以及执行定时作业。

(5) 第 10~20 行初始化集合。

getInitialActivityStack 方法主要用于将开始节点的表示类 ActivityImpl 加入到 ProcessDefinitionImpl 类中的 initialActivityStacks 集合,方便后续取值操作,然后将 initial 对象设置到 InterpretableExecution 实例对象中,方便后续流程实例启动时获取开始节点信息。

13.3.2 获取 dataObject

getVariables()方法用于获取流程文档中定义的 dataObject 元素信息,dataObject 元素的定义如代码清单 13-13 所示。

代码清单 13-13 dataObject.bpmn

```
1 <process id="dataObject" name="dataObject" isExecutable="true">
2   <dataObject id="shareniu1" name="longshateniu" itemSubjectRef="xsd:long"/>
3   <dataObject id="dObj123" name="shareniu" itemSubjectRef="xsd:string">
4     <extensionElements>
5       <activiti:value>shareniu123</activiti:value>
6     </extensionElements>
7   </dataObject>
8 </process>
```

Activiti 允许开发人员在流程文档中为流程或者子流程定义 dataObject 元素,该元素可以指定变量的 id、名称、数据类型等,支持的数据类型有 string、boolean、datetime、double、int、long 等。流程实例启动时会将 dataObject 元素的信息自动转换为流程实例变量存在,变量名称对应 dataObject 元素中定义的“name”值,开发人员也可以通过 dataObject 元素中的子元素 extensionElements 为转换之后的流程变量提供默认值。

在上述代码中,第 2 行定义了 id 为 shareniu1,name 为 longshateniu 的变量,第 3~7 行定义了 id 为 dObj123,name 为 shareniu 的变量,并在第 5 行为该变量设置默认值 shareniu123。

13.3.3 区别流程实例与执行实例

processInstance.setVariables(dataObjectVars)方法主要用于设置变量,变量设置的方式如代码清单 13-14 所示。

代码清单 13-14 RuntimeService.java

```
1 runtimeService.startProcessInstanceId(processDefinitionId, variables);
2 runtimeService.setVariables(executionId, variables);
3 runtimeService.setVariableLocal(executionId, variables);
4 taskService.complete(taskId, variables);
5 taskService.setVariableLocal(taskId, variables)
```

第 1 行在流程实例启动时设置变量(全局变量),第 2 行负责为流程实例设置变量,第 3 行负责为执行实例设置变量(局部变量),第 4 行任务完成时设置变量(全局变量),第 5 行为当前执行的任务设置局部变量。

了解了变量的设置方式之后,接下来讲解流程实例与执行实例的关系,通常情况下如果流程不存在分支或者多实例节点,则流程实例的 id 与执行实例的 id 相同,两者的值分别对

应 ACT_RU_EXECUTION 表中的 PROC_INST_ID 和 ID 列,对于存在分支或者多实例节点的场景可以结合图 13-2 进行理解。

接下来,分析一下 setVariables 和 setVariablesLocal 方法的区别。

setVariables 方式设置的变量都是针对流程实例的(全局变量),也就是上图中的二级节点(上图中自上而下分别对应一级节点、二级节点、三级节点)。

setVariablesLocal 方式设置的变量可以针对流程实例分支(局部变量),即执行实例,对应图 13-2 中的三级节点。

默认情况下变量都是针对流程实例的,而局部变量使用范围就比较小(与执行实例的生命周期绑定),只能针对执行实例,执行实例结束之后,后续的流程实例无法通过 ACT_RU_VARIABLE 表获取该变量值,只可以通过查询 ACT_HI_VARINST 表获取该变量值。

使用局部变量的好处,执行实例与流程实例可以使用相同的变量名称,而且相互之间不会冲突。使用全局变量,则变量名称相同时,后者会替换前者。

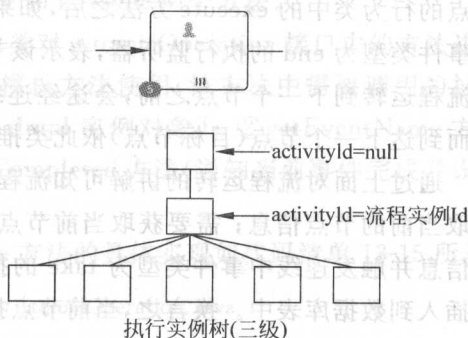


图 13-2 多实例任务执行 id 与流程实例 id 树

建议

可以启动一个附有多实例节点的流程实例对变量的不同设置方式进行操作,以加深理解。

13.3.4 添加历史流程实例数据

从数据库表设计思想上看,Activiti 将表划分为运行表和历史表从而大大减少运行表中的数据量,提高运行表的查询效率。Context.getCommandContext().getHistoryManager().recordProcessInstanceStart(processInstance)方法的职责就是将流程实例数据复制一份然后将其添加到会话缓存,引擎最终将该会话缓存中的数据添加到 ACT_HI_PROCINST 表中。

13.4 虚拟机运转原理

上文详细讲解了 ExecutionEntity 实例对象的创建过程,接下来讲解如何启动该实例对象。在学习之前先分析流程运转的设计思想,同样还是以图 11-1 为例进行讲解。

流程实例启动时,Activiti 会判断当前流程文档中的 process 元素是否配置有事件类型为 start 的执行监听器,如果有则会立即执行,执行完毕,Activiti 会找到流程文档的源头也就是开始节点,然后触发开始节点中事件类型为 start 的执行监听器,在第 10.5.1 节中讲解过创建 ActivityImpl 实例对象时会根据当前节点或者网关的类型为其添加不同的行为类,即执行完开始节点中事件类型为 start 的执行监听器之后,会执行开始节点的活动行为类中的 execute 方法,该方法决定了流程实例的最终走向,例如流程实例是否可以离开当前节点,途经哪些连线,最终流转到何方,均由开始节点的行为类进行干预控制,程序执行完开始

节点的行为类中的 `execute` 方法之后,如果流程实例可以继续向下运转则会执行当前节点中事件类型为 `end` 的执行监听器,表示该节点已经执行结束,流程实例可以继续向下流转,在流程运转到下一个节点之前,会途经连线并触发连线中事件类型为 `take` 的执行监听器,进而到达下一个节点(目标节点)依此类推。

通过上面对流程运转的讲解可知流程实例的运转过程中需要处理如下几个信息:需要获取当前的节点信息;需要获取当前节点的活动行为类;需要获取当前节点可以途经的连线信息并触发连线中事件类型为 `take` 的执行监听器;途经连线之后需要将目标节点的信息插入到数据库表中。换言之,当前节点执行完一系列的操作后会告诉引擎最终可以途径的连线,途径连线则会到达连线的目标节点,到达目标节点并执行完一系列的操作后,流程实例有可能如上述所示继续运转,直到流程实例结束为止。对于该处理过程而言,处理的节点不同,但是处理步骤却是相同的,通常这样的应用场景使用职责链模式最合适不过了,接下来分析 Activiti 是如何设计处理的。

13.5 AtomicOperation 架构

AtomicOperation 顾名思义原子性操作,是 Activiti 中的原子操作接口,该接口在流程虚拟机中占有非常重要的地位,负责整个流程实例运转调度工作,首先看一下 AtomicOperation 接口的架构,如图 13-3 所示。

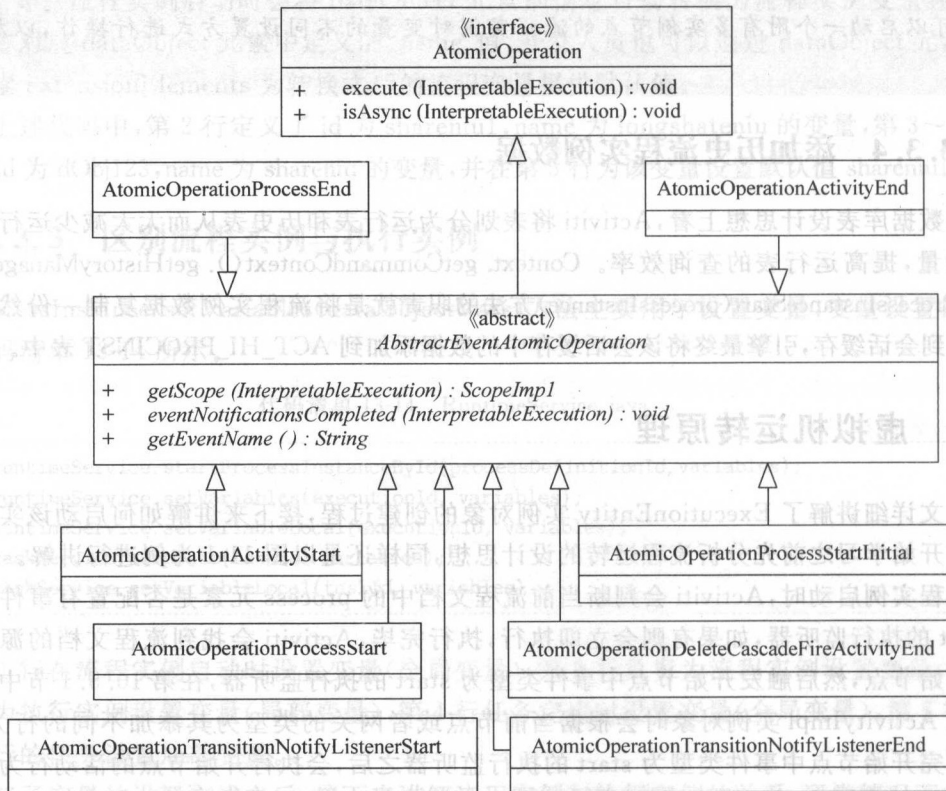


图 13-3 AtomicOperation 接口的架构图

(1) AtomicOperation: 该接口定义了 execute 和 isAsync 两个方法。

(2) AbstractEventAtomicOperation: 该抽象类对 AtomicOperation 接口中的方法进行实现, 在实现 execute 方法的同时, 将该方法作为模板方法使用, 该方法中需要调用的抽象方法包括如下三个: ① getScope 方法 (获取 ScopeImpl 实例对象); ② getEventName 方法 (获取监听器的事件类型); ③ eventNotificationsCompleted 方法 (通知当前事件完成并设置下一个原子类)。

AbstractEventAtomicOperation 类中 execute 方法的具体实现如代码清单 13-15 所示。

代码清单 13-15 AbstractEventAtomicOperation.java

```
1 public void execute(InterpretableExecution execution) {
2     ScopeImpl scope = getScope(execution);
3     List<ExecutionListener> executionListeners = scope.getExecutionListeners(getEventName());
4     int executionListenerIndex = execution.getExecutionListenerIndex();
5     if (executionListeners.size() > executionListenerIndex) {
6         execution.setEventName(getEventName());
7         execution.setEventSource(scope);
8         ExecutionListener listener = executionListeners.get(executionListenerIndex);
9         try {
10             listener.notify(execution);
11         } catch (RuntimeException e) {
12             throw e;
13         } catch (Exception e) {
14             throw new PvmException("couldn't execute event listener : " + e.getMessage(), e);
15         }
16         execution.setExecutionListenerIndex(executionListenerIndex + 1);
17         execution.performOperation(this);
18     } else {
19         execution.setExecutionListenerIndex(0);
20         execution.setEventName(null);
21         execution.setEventSource(null);
22         eventNotificationsCompleted(execution);
23     }
24 }
```

在上述代码中, 第 2 行调用 getScope 抽象方法获取 ScopeImpl 实例对象, 该方法的具体实现完全交给子类, 第 3 行代码执行了两步操作: ①通过 getEventName 方法获取执行监听器的事件类型, 例如 start 或者 end; ②获取步骤(1)中指定事件的执行监听器集合。

第 4 行中的 executionListenerIndex 变量有何意义呢? 仅仅在第 16 行代码中看到对该变量的赋值操作, 其实很容易理解, 例如在开始节点中定义了两个事件类型为 start 的执行监听器, 则通过第 3 行代码可以获取这两个执行监听器集合 (前提 getEventName() 返回值为 start), 这样第 5 行代码第一次执行的时候 executionListenerIndex 值为 0, executionListeners.size() 值为 2, 该代码执行完毕, 第 16 行会将 executionListenerIndex 的值在当前值的基础之上加 1, 从而保证所有的执行监听器都可以执行。

因为开发人员配置的执行监听器或者系统内置监听器均需要实现 ExecutionListener 接口并实现该接口中定义的 notify 方法, 所以第 10 行代码会触发执行监听器中的 notify

根据上面代码的执行逻辑可知, `execute` 方法的处理逻辑以执行监听器的有无为分水岭, 将执行监听器和非执行监听器的处理逻辑分开, 如果存在执行监听器则会直接调用第 17 行代码并将 `this` 指针传递过去, 如果当前处理类调用 `execute` 方法的同时需要处理执行监听器, 则处理完执行监听器后会再次执行该类中的 `execute` 方法, 直到所有的执行监听器处理完毕才会设置下一个需要处理的原子类; 如果没有执行监听器或者所有的执行监听器执行完毕则执行第 19~22 行代码, 其中 `eventNotificationsCompleted` 方法的职责就是设置下一步需要处理的原子类。

理解了上面的内容之后,接下来再次分析 `processInstance.start()` 的处理逻辑,该操作是流程启动的核心,相关实现如代码清单 13-16 所示。

```

1 public void start() {
2     if(startingExecution == null && isProcessInstanceType()) {
3         startingExecution = new StartingExecution(processDefinition.getInitial());
4     }
5     performOperation(AtomicOperation.PROCESS_START);
6 }

```

第5行委托 performOperation(AtomicOperation, PROCESS_START)方法进行流程实例的运转工作,该方法的定义如代码清单 13-17 所示。

```

1 public void performOperation(AtomicOperation executionOperation) {
2     if (executionOperation.isAsync(this)) {
3         scheduleAtomicOperationAsync(executionOperation);
4     } else {
5         performOperationSync(executionOperation);
6     }
7 }

```

第 2~6 行根据 `executionOperation.isAsync(this)` 判断程序是否需要异步处理,开发人员可以在流程文档中为节点定义是否需要异步处理属性,形如 `activiti:async="true"`,如果需要异步处理则第 3 行委托 `scheduleAtomicOperationAsync` 方法进行处理;否则第 5


```

10     } else {
11         currentOperation.execute(execution.getReplacedBy());
12     }
13 }
14 } finally {
15     Context.removeExecutionContext();
16 }
17 }
18 }

```

上面代码的处理流程非常清晰,第 2 行将 executionOperation 参数值添加到 nextOperations 集合中,第 3 行判断 nextOperations 集合的长度,如果该集合只有一个元素则执行下面的代码,否则不予处理。

第 5 行将 execution 参数值添加到 Context 类中,以方便程序后续获取。

第 7 行从 nextOperations 集合取出第一个元素,第 8 行判断 execution 对象中的 replacedBy 属性是否有值,不管 replacedBy 属性是否有值(流程实例通常作为执行实例的 parent 存在),最终都会调用原子类 AtomicOperation 的 execute 方法,而该方法在处理当前逻辑的同时会设置下一个需要处理的类,从而保证职责链中的类依次执行。第 15 行代码的操作与第 5 行操作完全相反。

13.6.2 异步节点处理

executionOperation.isAsync(this) 方法的返回值为 true,则调用 scheduleAtomicOperationAsync(executionOperation) 方法进行异步处理的相关实现如代码清单 13-20 所示。

代码清单 13-20 ExecutionEntity.java

```

1 protected void scheduleAtomicOperationAsync(AtomicOperation executionOperation) {
2     MessageEntity message = new MessageEntity();
3     message.setExecution(this);
4     message.setExclusive(getActivity().isExclusive());
5     message.setJobHandlerType(AsyncContinuationJobHandler.TYPE);
6     GregorianCalendar expireCal = new GregorianCalendar();
7     ProcessEngineConfiguration processEngineConfig =
8     Context.getCommandContext().getProcessEngineConfiguration();
9     expireCal.setTime(processEngineConfig.getClock().getCurrentTime());
10    expireCal.add(Calendar.SECOND, processEngineConfig.getLockTimeAsyncJobWaitTime());
11    message.setLockExpirationTime(expireCal.getTime());
12    if (getTenantId() != null) {
13        message.setTenantId(getTenantId());
14    }
15    Context.getCommandContext().getJobEntityManager().send(message);
16 }

```

该方法的处理逻辑如下。

(1) 第2行实例化 MessageEntity 类,该类的父类是 JobEntity(ACT_RU_JOB 表的实体类)。

(2) 第3行调用 message 对象的 setExecution 方法为其填充属性值,该方法的相关实现如代码清单 13-21 所示。

代码清单 13-21 JobEntity.java

```
1 public void setExecution(ExecutionEntity execution) {
2     executionId = execution.getId();
3     processInstanceId = execution.getProcessInstanceId();
4     processDefinitionId = execution.getProcessDefinitionId();
5     execution.addJob(this);
6 }
```

第2~4行分别填充当前类中的 executionId、processInstanceId、processDefinitionId 三个属性值,然后第5行调用 execution.addJob(this)方法,该方法的相关实现如代码清单 13-22 所示。

代码清单 13-22 JobEntity.java

```
1 public void addJob(JobEntity jobEntity) {
2     getJobsInternal().add(jobEntity);
3 }
```

addJob 方法直接调用 getJobsInternal()方法获取 List<JobEntity>集合,然后将 jobEntity 添加到该集合中,getJobsInternal()方法的相关实现代码清单 13-23 所示。

代码清单 13-23 JobEntity.java

```
1 protected List<JobEntity> jobs;
2 protected List<JobEntity> getJobsInternal() {
3     ensureJobsInitialized();
4     return jobs;
5 }
6 protected void ensureJobsInitialized() {
7     if(jobs == null) {
8         jobs = (List)Context.getCommandContext().getJobEntityManager().findJobsByExecutionId(id);
9     }
10 }
```

getJobsInternal 方法在第3行直接调用 ensureJobsInitialized 方法,然后第4行返回 jobs 集合,其中 ensureJobsInitialized 方法就是根据 id 值(执行 id)从 ACT_RU_JOB 表中查询数据。

(3) 第9~10行获取流程引擎配置类中的 lockTimeAsyncJobWaitTime 开关属性值,该值默认为60。

(4) 第15行调用 JobEntityManager 实例对象中的 send 方法进行处理,该方法的具体

实现如代码清单 13-24 所示。

代码清单 13-24 JobEntityManager.java

```

1 public void send(MessageEntity message) {
2     ProcessEngineConfigurationImpl processEngineConfiguration =
3     Context.getProcessEngineConfiguration();
4     if(processEngineConfiguration.isAsyncExecutorEnabled()) {
5         Date dueDate = new Date(processEngineConfiguration.getClock().getCurrentTime().getTime()
6         + processEngineConfiguration.getAsyncExecutor().getAsyncJobLockTimeInMillis());
7         message.setDuedate(dueDate);
8         message.setLockExpirationTime(null);
9     } else if (!processEngineConfiguration.isJobExecutorActivate()) {
10        message.setDuedate(processEngineConfiguration.getClock().getCurrentTime());
11        message.setLockExpirationTime(null);
12    }
13    message.insert();
14    if (processEngineConfiguration.isAsyncExecutorEnabled()) {
15        hintAsyncExecutor(message);
16    } else {
17        hintJobExecutor(message);
18    }
19 }

```

(1) 第 2~3 行通过 Context 类获取 ProcessEngineConfigurationImpl 实例对象。

(2) 第 4 行获取 processEngineConfiguration 对象中的 asyncExecutorEnabled 开关属性值,如果该属性值为 true,则实例化 Date 类,生成的日期=当前时间+ProcessEngineConfigurationImpl 类中的 asyncExecutorAsyncJobLockTimeInMillis 开关属性值,该开关属性值默认为 5min。通过该操作可知异步作业执行器优先级最高,如果开发人员开启了异步作业执行器功能,那么异步节点默认到期时间为 5min。

(3) 第 9 行获取 processEngineConfiguration 对象中的 jobExecutorActivate 开关属性值,如果该属性值为 false,则执行第 10~11 行代码。

(4) 第 13 行调用 message 对象的 insert 方法,该方法的相关实现如代码清单 13-25 所示。

代码清单 13-25 JobEntity.java

```

1 public void insert() {
2     Context.getCommandContext().getDbSqlSession().insert(this);
3     if (executionId != null) {
4         ExecutionEntity execution = Context.getCommandContext()
5         .getExecutionEntityManager().findExecutionById(executionId);
6         execution.addJob(this);
7         if (execution.getTenantId() != null) {
8             setTenantId(execution.getTenantId());
9         }
10    }
11    ...//转发 ENTITY_INITIALIZED 时间
12 }

```

在上述代码中,第2行将当前对象(JobEntity实例对象)添加到会话缓存中,第3行判断 executionId 是否为空,如果不为空,则第4~5行以 executionId 值为查询条件从 ACT_RU_EXECUTION 表中查询数据。

注意

上述代码中的 this 为 JobEntity 实例对象。

(5) 第15行调用 hintAsyncExecutor(message) 方法,该方法的具体实现如代码清单 13-26 所示。

代码清单 13-26 JobEntityManager.java

```
1 protected void hintAsyncExecutor(JobEntity job) {
2     AsyncExecutor asyncExecutor = Context.getProcessEngineConfiguration().getAsyncExecutor();
3     TransactionListener transactionListener = new AsyncJobAddedNotification(job, asyncExecutor);
4     Context.getCommandContext().getTransactionContext().
5     .addTransactionListener(TransactionState.COMMITTED, transactionListener);
6 }
```

在上述代码中,第2行获取 AsyncExecutor 实例对象,第3行实例化 TransactionListener 类,第4~5行将 transactionListener 对象添加到 TransactionContext 实例对象中。

通过上文的讲解可知对于异步节点的处理而言,优先使用异步作业执行器,其次是作业执行器。

13.7 原子类流转

13.7.1 流程启动原子类

本节以流程启动原子类 AtomicOperationProcessStart 为入口,探究流程实例运转过程中是如何使用职责链。AtomicOperationProcessStart 类的定义如代码清单 13-27 所示。

代码清单 13-27 AtomicOperationProcessStart.java

```
1 protected void eventNotificationsCompleted(InterpretableExecution execution) {
2     ...//省略转发转发 ENTITY_INITIALIZED 事件
3     ProcessDefinitionImpl processDefinition = execution.getProcessDefinition();
4     StartingExecution startingExecution = execution.getStartingExecution();
5     List<ActivityImpl> initialActivityStack =
6     processDefinition.getInitialActivityStack(startingExecution.getInitial());
7     execution.setActivity(initialActivityStack.get(0));
8     execution.performOperation(PROCESS_START_INITIAL);
9 }
```

将 eventNotificationsCompleted 方法的处理逻辑总结如下。

(1) 第3行获取 ProcessDefinitionImpl 实例对象。

(2) 第 4 行获取 StartingExecution 实例对象。StartingExecution 类中持有开始节点在流程虚拟机中的表示类 ActivityImpl。

(3) 第 5~6 行确保开始节点已经存储到 processDefinition 对象中。

(4) 第 7 行将开始节点存储到 execution 对象中,方便后续操作可以获取上一个已经执行的 ActivityImpl 实例对象。

(5) 第 8 行设置下一个需要处理的原子类 AtomicOperationProcessStartInitial。

13.7.2 流程启动准备原子类

在上面代码中指定的下一个原子类为 AtomicOperationProcessStartInitial,该类的定义如代码清单 13-28 所示。

代码清单 13-28 AtomicOperationProcessStartInitial.java

```
1 protected void eventNotificationsCompleted(InterpretableExecution execution) {
2     ActivityImpl activity = (ActivityImpl) execution.getActivity();
3     ProcessDefinitionImpl processDefinition = execution.getProcessDefinition();
4     StartingExecution startingExecution = execution.getStartingExecution();
5     if (activity == startingExecution.getInitial()) {
6         execution.disposeStartingExecution();
7         execution.performOperation(ACTIVITY_EXECUTE);
8     } else {
9         List<ActivityImpl> initialActivityStack =
10             processDefinition.getInitialActivityStack(startingExecution.getInitial());
11         int index = initialActivityStack.indexOf(activity);
12         activity = initialActivityStack.get(index + 1);
13         InterpretableExecution executionToUse = null;
14         if (activity.isScope()) {
15             executionToUse = (InterpretableExecution) execution.getExecutions().get(0);
16         } else {
17             executionToUse = execution;
18         }
19         executionToUse.setActivity(activity);
20         executionToUse.performOperation(PROCESS_START_INITIAL);
21     }
22 }
```

上述代码的处理逻辑总结如下。

(1) 第 2 行通过 execution 对象的 getActivity 方法获取 ActivityImpl 实例对象(对应开始节点的信息)。

(2) 第 4 行通过 execution 对象中的 getStartingExecution 方法获取 StartingExecution 实例对象,

(3) 第 5 行判断 activity 与 startingExecution.getInitial() 方法的返回值是否相同,如果相同则首先第 6 行调用 execution.disposeStartingExecution() 方法将 execution 对象中的 startingExecution 属性值设置为空,然后第 7 行设置下一个需要处理的原子类 AtomicOperationActivityExecute; 如果不相同则执行第 9~20 行代码,需要重新获取开始节点的

信息,并设置下一个需要处理的原子类为 `AtomicOperationProcessStartInitial`,也就是当前类。该操作主要是为了确保流程实例的启动节点与流程文档中定义的开始节点为同一个节点。

13.7.3 活动节点执行原子类

`AtomicOperationActivityExecute` 类中 `execute` 方法的相关实现如代码清单 13-29 所示。

代码清单 13-29 `AtomicOperationActivityExecute.java`

```
1 public void execute(InterpretableExecution execution) {
2     ActivityImpl activity = (ActivityImpl) execution.getActivity();
3     ActivityBehavior activityBehavior = activity.getActivityBehavior();
4     if (activityBehavior == null) {
5         throw new PvmException("no behavior specified in " + activity);
6     }
7     try {
8         ...//省略转发 ACTIVITY_STARTED 事件
9         activityBehavior.execute(execution);
10    } catch (Exception e) {
11        throw new PvmException("couldn't execute activity " + activity.getProperty("type"));
12    }
13 }
```

在上述代码中,第 2 行通过 `execution` 对象的 `getActivity` 方法获取 `ActivityImpl` 实例对象,然后第 3 行根据该实例对象获取开始节点对应的行为类实例对象 `activityBehavior` (该实例对象为 `NoneStartEventActivityBehavior` 类型),并在第 9 行执行该实例对象中的 `execute` 方法。

13.7.4 开始节点行为类

接下来,分析 `NoneStartEventActivityBehavior` 类中的 `execute` 方法处理逻辑,`NoneStartEventActivityBehavior` 类中并没有定义 `execute` 方法,因此需要跟进该类的父类 `FlowNodeActivityBehavior`,`FlowNodeActivityBehavior` 类中 `execute` 方法的相关实现如代码清单 13-30 所示。

代码清单 13-30 `FlowNodeActivityBehavior.java`

```
1 public void execute(ActivityExecution execution) throws Exception {
2     leave(execution);
3 }
```

第 2 行直接调用 `leave` 方法进行处理,`leave` 方法的定义如代码清单 13-31 所示。

代码清单 13-31 `FlowNodeActivityBehavior.java`

```
1 protected BpmnActivityBehavior bpmnActivityBehavior = new BpmnActivityBehavior();
```



```

2 protected void leave(ActivityExecution execution) {
3     bpmnActivityBehavior.performDefaultOutgoingBehavior(execution);
4 }

```

第 1 行实例化 BpmnActivityBehavior 类,第 3 行调用 bpmnActivityBehavior 对象的 performDefaultOutgoingBehavior 方法进行下一步的处理,performDefaultOutgoingBehavior 方法的实现如代码清单 13-32 所示。

代码清单 13-32 BpmnActivityBehavior.java

```

1 public void performDefaultOutgoingBehavior(ActivityExecution activityExecution) {
2     ActivityImpl activity = (ActivityImpl) activityExecution.getActivity();
3     if (!(activity.getActivityBehavior() instanceof IntermediateCatchEventActivityBehavior)) {
4         dispatchJobCanceledEvents(activityExecution);
5     }
6     performOutgoingBehavior(activityExecution, true, false, null);
7 }

```

第 2 行获取 ActivityImpl 实例对象,第 3 行如果获取的 activity 行为类实例对象不是 IntermediateCatchEventActivityBehavior 实例对象,则第 4 行调用 dispatchJobCanceledEvents 方法转发 JOB_CANCELED 事件,第 6 行开始真正执行活动行为类,该方法的实现如代码清单 13-33 所示。

代码清单 13-33 BpmnActivityBehavior.java

```

1 protected void performOutgoingBehavior(ActivityExecution execution,
2     boolean checkConditions, boolean throwExceptionIfExecutionStuck, List reusableExecutions) {
3     String defaultSequenceFlow = (String) execution.getActivity().getProperty("default");
4     List<PvmTransition> transitionsToTake = new ArrayList<PvmTransition>();
5     List<PvmTransition> outgoingTransitions = execution.getActivity().getOutgoingTransitions();
6     for (PvmTransition outgoingTransition : outgoingTransitions) {
7         Expression skipExpression = outgoingTransition.getSkipExpression();
8         if (!SkipExpressionUtil.isSkipExpressionEnabled(execution, skipExpression)) {
9             if (defaultSequenceFlow == null || !outgoingTransition.getId().equals(defaultSequenceFlow)) {
10                 Condition condition = (Condition)
11                     outgoingTransition.getProperty(BpmnParse.PROPERTYNAME_CONDITION);
12                 if (condition == null || !checkConditions ||
13                     condition.evaluate(outgoingTransition.getId(), execution)) {
14                     transitionsToTake.add(outgoingTransition);
15                 }
16             }
17             else if (SkipExpressionUtil.shouldSkipFlowElement(execution, skipExpression)) {
18                 transitionsToTake.add(outgoingTransition);
19             }
20         }
21         if (transitionsToTake.size() == 1) {
22             execution.take(transitionsToTake.get(0));

```

```

23 } else if (transitionsToTake.size() >= 1) {
24     execution.inactivate();
25     if (reusableExecutions == null || reusableExecutions.isEmpty()) {
26         execution.takeAll(transitionsToTake, Arrays.asList(execution));
27     } else {
28         execution.takeAll(transitionsToTake, reusableExecutions);
29     }
30 } else {
31     if (defaultSequenceFlow != null) {
32         PvmTransition defaultTransition =
33             execution.getActivity().findOutgoingTransition(defaultSequenceFlow);
34         if (defaultTransition != null) {
35             execution.take(defaultTransition);
36         } else {
37             throw new ActivitiException("Default sequence '" + defaultSequenceFlow + "' not
found");
38         }
39     } else {
40         Object isForCompensation =
41             execution.getActivity().getProperty(BpmnParse.PROPERTYNAME_IS_FOR_COMPENSATION);
42         if (isForCompensation != null && (Boolean) isForCompensation) {
43             if (execution instanceof ExecutionEntity) {
44                 Context.getCommandContext().getHistoryManager().recordActivityEnd((ExecutionEntity)
execution);
45             }
46         }
47         InterpretableExecution parentExecution = (InterpretableExecution) execution.
getParent();
48         ((InterpretableExecution) execution).remove();
49         parentExecution.signal("compensationDone", null);
50     } else {
51         execution.end();
52     }
53 }
54 }
55 }

```

上面代码的处理逻辑非常复杂,其中涉及了各种各样的考虑,如果细心阅读上面的代码,并参考部分代码注释,可以粗略地了解整个执行过程,下面对其进行细化讲解。

注意

execute 方法作为所有活动行为类的模板方法存在,因此该方法的处理逻辑比较复杂,需要考虑所有可能的情况,所以下面的讲解内容不要仅仅局限于 NoneStartEventActivityBehavior 类的处理。

- (1) 第 3 行获取当前节点的"default"属性值。
- (2) 第 5 行获取当前节点的所有出线信息并使用 outgoingTransitions 变量进行存储。
- (3) 第 6~18 行遍历所有的连线集合 outgoingTransitions。

如果连线配置有 skipExpression 属性,则首先第 8 行委托 SkipExpressionUtil 类验证

skipExpression 表达式的是否存在,如果存在,则第 17 行继续验证表达式是否为 true,如果是则第 18 行将其添加到 transitionsToTake 集合;如果连线中没有配置 skipExpression 属性值则开始执行如下的逻辑。

- 第 9 行如果 defaultSequenceFlow 变量值为空或者该变量不存在于 outgoingTransitions 集合中,则第 10~11 行获取 outgoingTransition 对象中的 condition 属性值并使用 condition 变量进行表示,经过第 12~13 行的验证操作,如果 condition 变量为空,或者 checkConditions 参数值为 false,或者条件表达式计算之后验证通过,则第 14 行将 outgoingTransition 对象添加到 transitionsToTake 集合。

根据上面的处理也可以看出,Activiti 中的节点本身就支持并行性操作,例如为开始节点配置三个出线,而三个出线均没有配置任何表达式,则流程实例运转过程都会途经这三个连线。

(4) 途经连线。

步骤(3)使用 transitionsToTake 集合存储节点可以途经的连线信息,然后对其进行处理,逻辑如下。

- 第 21~22 行如果 transitionsToTake 集合只存在一个元素则会执行 execution.take() 方法。
- 第 24~29 行如果该集合存在多个元素,则首先通过 execution.inactivate() 设置 ExecutionEntity 实例对象中的 isActive 属性值为 false,然后执行 execution.takeAll() 方法。
- 如果 transitionsToTake 集合为空,则首先第 31 行判断 defaultSequenceFlow(默认连线)变量是否为空,如果该变量不为空,则第 32~33 行需要获取该连线对应的流程虚拟机对象并使用 defaultTransition 变量进行存储,如果 defaultTransition 不为空,则第 35 行执行 execution.take(defaultTransition) 方法,否则程序报错;如果 defaultSequenceFlow 为空,则首先第 40~41 行查找当前节点是否配置有 isForCompensation 属性值,如果 isForCompensation 属性存在并且为 true,则第 49 行程序执行 parentExecution.signal("compensationDone", null) 操作,使用发射信号方式完成该活动节点,从而推动流程实例继续向下运转,如果 isForCompensation 属性不存在或者存在但该值为 false,则第 51 行直接结束当前的流程实例。

13.7.5 途经连线

transitionsToTake 集合中存在一个元素时,会委托 execution.take() 方法进行流程实例的运转工作,该方法的具体实现如代码清单 13-34 所示。

代码清单 13-34 ExecutionEntity.java

```
1 public void take(PvmTransition transition) {
2     take(transition, true);
3 }
```

第 2 行直接调用 take(transition, true) 方法,该方法的定义如代码清单 13-35 所示。

代码清单 13-35 ExecutionEntity.java

```
1 public void take(PvmTransition transition, boolean fireActivityCompletionEvent) {
2     if (fireActivityCompletionEvent) {
3         fireActivityCompletedEvent();
4     }
5     if (this.transition != null) {
6         throw new PvmException("already taking a transition");
7     }
8     if (transition == null) {
9         throw new PvmException("transition is null");
10    }
11    setActivity((ActivityImpl) transition.getSource());
12    setTransition((TransitionImpl) transition);
13    performOperation(AtomicOperation.TRANSITION_NOTIFY_LISTENER_END);
14 }
```

上述代码的处理逻辑进行如下总结。

- (1) 第2行如果 fireActivityCompletionEvent 参数值为 true, 则调用 fireActivityCompletedEvent 方法进行处理, 该方法主要用于转发 ACTIVITY_COMPLETED 事件。
- (2) 第5行校验 ExecutionEntity 实例对象中的 transition 属性值, 如果该属性值不为空, 则第6行程序直接报错。该属性用来存储已经执行过的连线。
- (3) 第8行进行 transition 参数的非空校验。
- (4) 第11~12行设置 ExecutionEntity 实例对象的属性值。
- (5) 第13行设置下一个需要处理的原子类 AtomicOperationTransitionNotifyListenerEnd。

13.7.6 通知连线完成原子类

AtomicOperationTransitionNotifyListenerEnd 类主要负责触发当前节点中配置的事件类型为 end 的执行监听器以及指定下一个需要处理的原子类 AtomicOperationTransitionDestroyScope, 该类的定义如代码清单 13-36 所示。

代码清单 13-36 AtomicOperationTransitionNotifyListenerEnd.java

```
1 protected String getEventName() {
2     return org.activiti.engine.impl.pvm.PvmEvent.EVENTNAME_END;
3 }
4 protected void eventNotificationsCompleted(InterpretableExecution execution) {
5     execution.performOperation(TRANSITION_DESTROY_SCOPE);
6 }
```

该类仅仅做了两个操作: 第一, 指定下一个需要处理的原子类 AtomicOperationTransitionDestroyScope; 第二, 触发连线中事件类型为 end 的执行监听器。

13.7.7 连线销毁原子类

AtomicOperationTransitionDestroyScope 类直接实现了 AtomicOperation 接口, 该类

中 execute 方法的具体实现如代码清单 13-37 所示。

代码清单 13-37 AtomicOperationTransitionDestroyScope.java

```

1 public void execute(InterpretableExecution execution) {
2     InterpretableExecution propagatingExecution = null;
3     ActivityImpl activity = (ActivityImpl) execution.getActivity();
4     if (activity.isScope()) {
5         ...//省略分支执行逻辑
6     } else {
7         propagatingExecution = execution;
8     }
9     ScopeImpl nextOuterScopeElement = activity.getParent();
10    TransitionImpl transition = propagatingExecution.getTransition();
11    ActivityImpl destination = transition.getDestination();
12    if (transitionLeavesNextOuterScope(nextOuterScopeElement, destination)) {
13        propagatingExecution.setActivity((ActivityImpl) nextOuterScopeElement);
14        propagatingExecution.performOperation(TRANSITION_NOTIFY_LISTENER_END);
15    } else {
16        propagatingExecution.performOperation(TRANSITION_NOTIFY_LISTENER_TAKE);
17    }
18 }

```

该方法的处理逻辑比较复杂,首先第 4 行调用 activity.isScope()方法,此方法的返回值主要记录引用流程 CallActivity、Transaction 元素、TimerEventDefinition 等,因为这些对象的处理与常规对象的处理有差异,最简单的理解就是引用流程或者多实例节点在流程实例运转的过程中均存在执行实例和流程实例的概念,因此需要使用 isScope 属性进行标示。可以使用 eclipse 工具打开 ActivityImpl 类,然后将光标定位到 setScope 方法,最后单击右键 References、WorkSpace 面板查看哪些元素支持该属性,最终查看效果如图 13-4 所示。

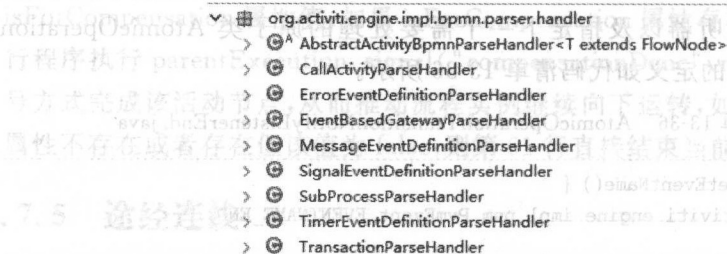


图 13-4 isScope 的引用类

理解了 activity.isScope()方法返回值的含义之后,接下来将重心放到非多实例节点的运转中,第 9 行获取 activity 对象中的 parent 属性值,第 10 行获取当前节点的连线信息,第 11 行通过 transition 获取当前节点的目标节点也就是流程实例最终需要运转到的节点。如果 nextOuterScopeElement 对象中存在目标节点,则第 16 行直接设置下一个需要处理的原子类 AtomicOperationTransitionNotifyListenerTake,否则第 14 行设置下一个需要处理的原子类为 AtomicOperationTransitionNotifyListenerEnd。

13.7.8 其他原子类

(1) AtomicOperationTransitionNotifyListenerTake: 该类主要负责触发连线中类型为 take 的执行监听器, 获取和设置当前节点的目标节点信息, 并设置下一个需要处理的原子类 AtomicOperationTransitionCreateScope。

(2) AtomicOperationTransitionCreateScope: 该类负责设置下一个需要处理的原子类 AtomicOperationTransitionNotifyListenerStart。

(3) AtomicOperationTransitionNotifyListenerStart: 该类负责查找当前节点需要到达的目标节点以及触发目标节点中类型为 start 的执行监听器。

这里简单说明一下, 流程实例到达目标节点之后, 首先要做的工作是创建目标节点以及将该节点的数据入库, 创建目标节点之后, 如果目标节点不能推动流程实例继续向下运转, 则其就是原子类结束的标志。

建议

可以一直跟进原子类, 最终可以发现如果当前原子类没有设置下一个需要处理的原子类则意味着当前操作已经结束。

13.8 Activiti 新特性之忽略节点

skipExpression 属性为 Activiti 5.17 版本的新特性, 该操作用来设置流程实例运转过程中是否可以忽略当前的活动节点, 以 SkipExpressionUtil.isSkipExpressionEnabled(execution, skipExpression) 这行代码为切入口进行分析, 相关实现如代码清单 13-38 所示。

代码清单 13-38 SkipExpressionUtil.java

```
1 public static boolean isSkipExpressionEnabled(ActivityExecution execution, Expression
   skipExpression)
2 {
3     if (skipExpression == null) {
4         return false;
5     }
6     final String skipExpressionEnabledVariable = "_ACTIVITI_SKIP_EXPRESSION_ENABLED";
7     Object isSkipExpressionEnabled = execution.getVariable(skipExpressionEnabledVariable);
8     if (isSkipExpressionEnabled == null) {
9         return false;
10    } else if (isSkipExpressionEnabled instanceof Boolean) {
11        return ((Boolean) isSkipExpressionEnabled).booleanValue();
12    } else {
13        throw new ActivitiIllegalArgumentException(skipExpressionEnabledVariable);
14    }
15 }
```

将 isSkipExpressionEnabled 方法的处理逻辑理总结如下。

(1) 第 3 行对 skipExpression 参数值进行非空校验, 如果该参数值为空则第 4 行直接返回 false。

(2) 第 7 行获取名称为_ACTIVITI_SKIP_EXPRESSION_ENABLED 的变量值, 如果该变量值为空, 则第 9 行直接返回 false。

(3) 第 10 行如果 isSkipExpressionEnabled 值为 Boolean 类型, 则第 11 行对其进行转化并将转化后的结果作为该方法的返回值。

(4) 如果 isSkipExpressionEnabled 值为非 Boolean 类型, 则程序报错。

通过 isSkipExpressionEnabled 方法的处理逻辑可以看出要想使用忽略节点功能, 则变量的名称必须为_ACTIVITI_SKIP_EXPRESSION_ENABLED 并且为 Boolean 类型, 该功能的配置形如代码清单 13-39 所示。

代码清单 13-39 skipExpression.bpmn

```

1 <process id="skipExpression" name="skipExpression" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="usertask1"
4     activiti:skipExpression="$ {_ACTIVITI_SKIP_EXPRESSION_ENABLED}"></userTask>
5   <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
6   <endEvent id="endevent1" name="End"></endEvent>
7   <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="endevent1"></sequenceFlow>
8 </process>

```

在上述代码中, 第 3~4 行为 id 为 usertask1 的任务节点配置了 activiti:skipExpression 属性值, 接下来的工作就是部署该流程文档并启动一个新的流程实例, 启动流程实例的同时需要设置变量如代码清单 13-40 所示。

代码清单 13-40 App.java

```

1 public void startProcessInstanceById() {
2   Map<String, Object> map = new HashMap<String, Object>();
3   map.put("_ACTIVITI_SKIP_EXPRESSION_ENABLED", true);
4   runtimeService.startProcessInstanceById("skipExpression:2:25003", map);
5 }

```

按照上述的步骤进行操作, 如果忽略节点功能生效则新启动的流程实例直接结束。关于忽略节点更多的使用技巧可以参考第 14.6 节。

在前面章节中多次提到创建 ActivityImpl 实例对象过程中,会根据节点的类型注入不同的活动行为类,那么活动行为类主要用于完成什么功能?通俗一点描述,行为就是受大脑中的想法或意识支配完成一系列的动作,例如现在需要完成一个任务,对于任务节点的执行行为来说,需要考虑如下几种情况:当前任务是否满足结束条件,任务完成之后流程实例可以途径的连线,最终到达的目的地等。换言之,任务节点的行为类决定了该流程实例的最终走向。活动行为类在流程虚拟机中占有非常重要的地位,所以本章会重点讲解,希望从中有所收获。

14.1 活动行为工厂类

Activiti 将所有活动行为类的创建工作交给活动行为工厂类完成,因此在讲解活动行为类之前,首先讲解活动行为工厂类的相关知识。

14.1.1 初始化活动行为工厂类

7.1 节中 getDefaultDeployers 方法用于实例化活动行为工厂类,即 DefaultActivityBehaviorFactory 类,详细过程如代码清单 14-1 所示。

代码清单 14-1 ProcessEngineConfigurationImpl.java

```
1 protected ActivityBehaviorFactory activityBehaviorFactory;
2 protected Collection<? extends Deployer> getDefaultDeployers() {
3     ...//省略部署器的初始化过程
4     if (activityBehaviorFactory == null) {
5         DefaultActivityBehaviorFactory defaultActivityBehaviorFactory = new
6         DefaultActivityBehaviorFactory();
7         defaultActivityBehaviorFactory.setExpressionManager(expressionManager);
```



```

8      activityBehaviorFactory = defaultActivityBehaviorFactory;
9    }
10   ...//省略对象解析器初始化
11   }

```

上述代码中,第 4 行判断 `ProcessEngineConfigurationImpl` 类中的 `activityBehaviorFactory` 开关属性值是否为空,如果该属性值为空,则第 5~6 行实例化 `DefaultActivityBehaviorFactory`,然后第 7 行为 `defaultActivityBehaviorFactory` 对象填充表达式管理器属性值。通过上面的代码可以看出如果想要自定义活动行为工厂类,最简单的方法就是定义一个类并继承 `DefaultActivityBehaviorFactory` 类,然后设置 `activityBehaviorFactory` 开关属性即可。

14.1.2 活动行为类架构

`DefaultActivityBehaviorFactory` 类实现了 `ActivityBehaviorFactory` 接口的同时还继承了 `AbstractBehaviorFactory` 类, `AbstractBehaviorFactory` 类内部持有表达式管理器,主要负责将 `FieldExtension` 类型的集合转化为 `FieldDeclaration` 类型的集合(activiti:field 元素)。经历了前面章节的学习,或多或少会发现一个规律,通常情况下 Activiti 会将一些复杂对象的创建工作交给相应的工厂类,活动行为类的创建也不例外,所有活动行为类的创建方法均定义在 `ActivityBehaviorFactory` 接口中。

为何需要通过抽象工厂进行一系列实例对象的创建工作,而不是将每一个实例对象的创建方法分散到不同的类中呢?其实在同一个地方集中创建一系列实例对象的好处就是可以对创建的实例对象集中管理,也方便对抽象工厂类进行维护,附带的好处就是当开发人员需要扩展活动行为工厂类的默认实现时,只需在同一个地方进行操作即可,从而降低客户端使用的复杂度,例如需要扩展某个活动行为类的默认实现时,只需自定义一个类并继承 `DefaultActivityBehaviorFactory` 类,覆盖该活动行为工厂类的创建方法即可。

`ActivityBehaviorFactory` 接口的默认实现类为 `DefaultActivityBehaviorFactory`, Activiti 通过 `DefaultActivityBehaviorFactory` 类衍生一系列的活动行为类,那么这些衍生的活动行为类分别负责什么工作呢?首先查看这些类的架构图如图 14-1 所示。

(1) `ActivityBehavior`: 该接口定义了 `execute` 方法,该方法决定了流程实例最终可以到达的目的地以及途径的连线。

(2) `SignallableActivityBehavior`: 在 `ActivityBehavior` 接口的基础上增加了执行任务的信号发射方法 `signal`, 形如 `runtimeService.signal(executionId)`, 其中 `executionId` 输入参数数值表示流程实例的执行 id 值。

(3) `FlowNodeActivityBehavior`: 该抽象类对 `SignallableActivityBehavior` 接口以及 `ActivityBehavior` 接口定义的方法进行实现,同时该类作为流程三要素的活动行为类的父类存在,这也意味着该类的任何子类可以是连线 `sequenceflow` 的源或目标。

(4) `AbstractBpmnActivityBehavior`: 该类是所有子流程、任务、引用流程的活动行为类的父类,并增加了对多实例活动行为类的支持。

(5) `TaskActivityBehavior`: 该类是所有任务行为类的父类,例如 `ServiceTask`、`ScriptTask`、`UserTask` 等,该类中没有定义任何方法,仅仅是为了约束任务行为类而已。

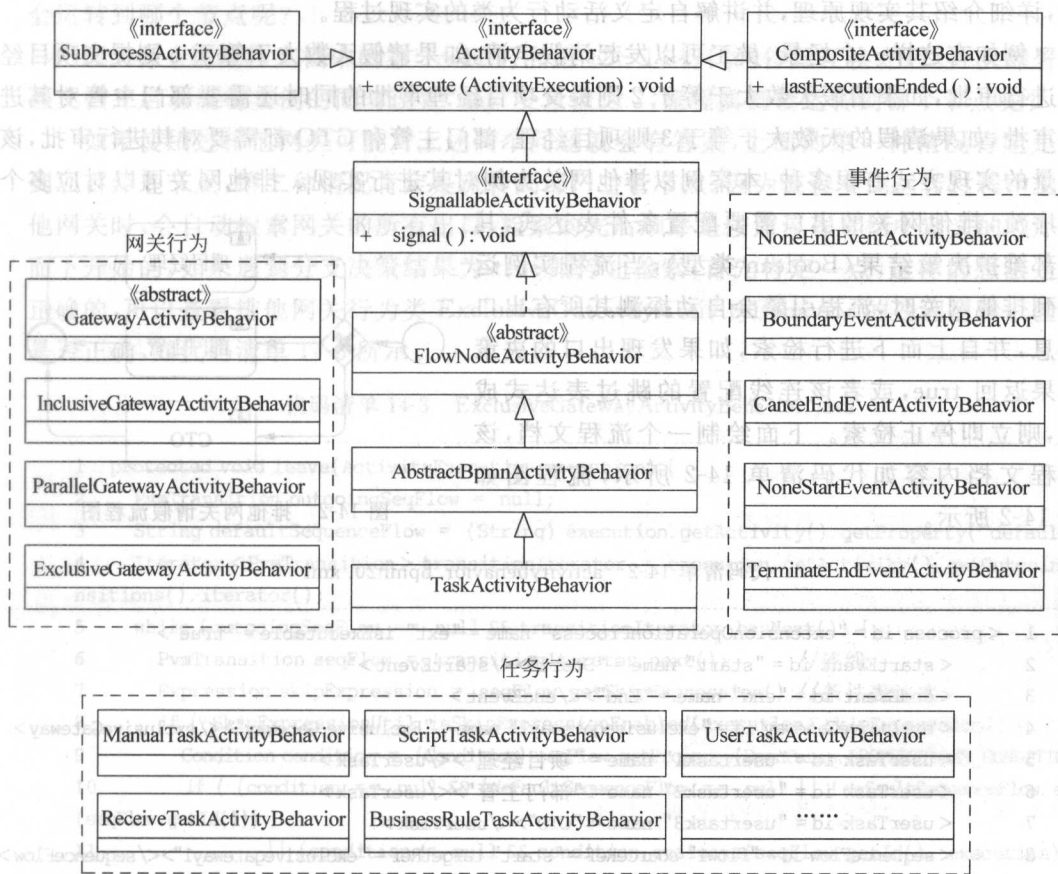


图 14-1 活动行为类架构

(6) SubProcessActivityBehavior: 该接口继承 ActivityBehavior 接口, 增加了 completing 和 completed 两个方法, completing 方法可以在子流程实例销毁之前从子流程中提取数据, 该方法供开发人员扩展使用, completed 方法控制流程实例是否可以继续向下运转。

(7) CompositeActivityBehavior: 该接口在 ActivityBehavior 接口基础之上增加了处理多实例或者子流程元素的方法 lastExecutionEnded, 例如多实例任务流转时, 流程实例永远只有一个, 但是执行实例可以存在多个, 而且流程实例作为所有执行实例的 parent 存在, 当多实例节点结束时需要根据当前的执行实例查询顶级 parent 信息也就是流程实例信息, 并最终决定流程实例运转的目的地, 所以多实例任务需要与非多实例任务要区别对待。

(8) GatewayActivityBehavior: 所有网关类型活动行为类的父类, 提供了 lockConcurrentRoot 方法供子类“包含网关”以及“并行网关”使用。

14.2 排他网关行为类原理

由于 Activiti 中活动行为类非常多, 每个类都进行讲解, 耗费精力太大, 而且可能有事倍功半的效果, 所以接下来以排他网关活动行为类 ExclusiveGatewayActivityBehavior 为

例,详细介绍其实现原理,并讲解自定义活动行为类的实现过程。

例如有这样一个场景,员工可以发起请假申请,如果请假天数大于等于 1 则提交项目经理进行审批,如果请假天数大于等于 2 则提交项目经理审批的同时还需要部门主管对其进行审批,如果请假的天数大于等于 3 则项目经理、部门主管和 CTO 都需要对其进行审批,该场景的实现方式有很多种,本案例以排他网关为例对其进行实现。排他网关可以对应多个顺序流,排他网关的出口需要配置条件表达式,其内部维护决策结果(Boolean 类型),当流程实例运转到排他网关时,流程引擎会自动探测其所有出口信息,并自上而下进行检索,如果发现出口的决策结果返回 true,或者该连线配置的跳过表达式成立,则立即停止检索。下面绘制一个流程文档,该流程文档内容如代码清单 14-2 所示,流程图如图 14-2 所示。

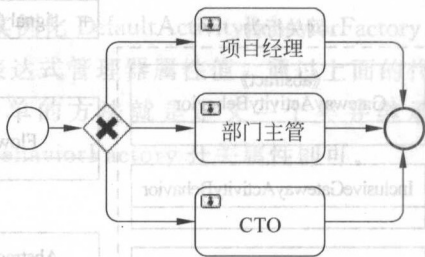


图 14-2 排他网关请假流程图

代码清单 14-2 activitybehavior.bpmn20.xml

```

1 <process id="extensionOperationProcess" name="ext" isExecutable="true">
2   <startEvent id="start" name="Start"></startEvent>
3   <endEvent id="end" name="End"></endEvent>
4   <exclusiveGateway id="exclusivegateway1" name="Exclusive Gateway"></exclusiveGateway>
5   <userTask id="usertask1" name="项目经理"></userTask>
6   <userTask id="usertask2" name="部门主管"></userTask>
7   <userTask id="usertask3" name="CTO"></userTask>
8   <sequenceFlow id="flow1" sourceRef="start" targetRef="exclusivegateway1"></sequenceFlow>
9   <sequenceFlow id="flow8" sourceRef="exclusivegateway1" targetRef="usertask1">
10    <conditionExpression xsi:type="tFormalExpression"><![CDATA[ ${ day >= 1 } ]]>
</conditionExpression>
11  </sequenceFlow>
12  <sequenceFlow id="flow3" sourceRef="exclusivegateway1" targetRef="usertask2">
13    <conditionExpression xsi:type="tFormalExpression"><![CDATA[ ${ day >= 2 } ]]>
</conditionExpression>
14  </sequenceFlow>
15  <sequenceFlow id="flow4" sourceRef="exclusivegateway1" targetRef="usertask3">
16    <conditionExpression xsi:type="tFormalExpression"><![CDATA[ ${ day >= 3 } ]]>
</conditionExpression>
17  </sequenceFlow>
18  <sequenceFlow id="flow5" sourceRef="usertask1" targetRef="end"></sequenceFlow>
19  <sequenceFlow id="flow6" sourceRef="usertask2" targetRef="end"></sequenceFlow>
20  <sequenceFlow id="flow7" sourceRef="usertask3" targetRef="end"></sequenceFlow>
21 </process>

```

根据上述代码可以看出,排他网关的分支条件必须在流程文档中进行定义,分支条件的配置比较简单,一般为 boolean 类型的表达式,第 9~11 行对应的连线目标节点为项目经理,第 12~14 行对应的连线目标节点为部门主管,第 15~17 行对应的连线目标节点为 CTO,接下来思考如下两个问题。

(1) 将上述流程文档进行部署并启动流程实例的同时设置“day”变量值为 2,流程实例

会运转到哪个节点呢？

(2) 将上述流程文档中的第 9~11 行和第 12~14 行代码位置互换之后，再次部署该流程文档并启动流程实例的同时设置“day”变量值为 2，流程实例会运转到哪个节点呢？

如果使用过排他网关可能对上述两个问题就会有答案，上面的第一种情况肯定是流转到项目经理节点，第二种情况肯定是流转到部门主管节点，因为流程实例运转过程中遇到排他网关时，会自动检索网关的所有出口，检索的先后顺序是按照流程文档中定义的顺序自上而下开始的，如果遇到分支决策结果为 true 则停止检索，首先肯定一点，这样的想法是完全正确的，可以查看排他网关行为类 ExclusiveGatewayActivityBehavior 的处理逻辑以验证其是否正确，如代码清单 14-3 所示。

14.3.2 自定义 代码清单 14-3 ExclusiveGatewayActivityBehavior.java

```

1  protected void leave(ActivityExecution execution) {
2      PvmTransition outgoingSeqFlow = null;
3      String defaultSequenceFlow = (String) execution.getActivity().getProperty("default");
4      Iterator<PvmTransition> transitionIterator = execution.getActivity().getOutgoingTransitions().iterator();
5      while (outgoingSeqFlow == null && transitionIterator.hasNext()) {
6          PvmTransition seqFlow = transitionIterator.next(); // 连线
7          Expression skipExpression = seqFlow.getSkipExpression(); // 条件表达式
8          if (!SkipExpressionUtil.isSkipExpressionEnabled(execution, skipExpression)) {
9              Condition condition = (Condition) seqFlow.getProperty(BpmnParse.PROPERTYNAME_CONDITION);
10             if ( (condition == null && (defaultSequenceFlow == null || !defaultSequenceFlow.equals(seqFlow.getId()))) ||
11                 (condition != null && condition.evaluate(seqFlow.getId(), execution)) ) {
12                 outgoingSeqFlow = seqFlow;
13             }
14         }
15         else if (SkipExpressionUtil.shouldSkipFlowElement(execution, skipExpression)) {
16             outgoingSeqFlow = seqFlow;
17         }
18     }
19     if (outgoingSeqFlow != null) {
20         execution.take(outgoingSeqFlow);
21     } else {
22         if (defaultSequenceFlow != null) {
23             PvmTransition defaultTransition = execution.getActivity().findOutgoingTransition(defaultSequenceFlow);
24             if (defaultTransition != null) {
25                 execution.take(defaultTransition);
26             } else {
27                 throw new ActivitiException("Default sequence flow not found");
28             }
29         } else {
30             throw new ActivitiException("No outgoing sequence flow of the exclusive gateway");
31         }
32     }
33 }

```


通过上面的代码可以看出,一个排他网关可以对应多个出口,而出口可以通过"condition"属性值进行设置,不管有多少个符合条件的出口,通过第 20 行的 execution.take(outgoingSeqFlow)操作发现排他网关只会执行一条出口,将其处理逻辑总结如下。

(1) 第 3 行获取排他网关中默认连线的 id 值,即"default"属性值。

(2) 第 4 行检索排他网关所有的出口。

(3) 第 5~18 行遍历所有的出口集合,第 15 行会查询当前的出口是否配置有“跳过表达式”,该属性的优先级非常高,如果当前被遍历的出口配置有该属性,则只要“跳过表达式”返回 true 就会立即停止顺序流的遍历工作,否则第 10~11 行会继续查询并验证连线表达式是否符合条件(包括没有配置条件表达式的出口),如果符合条件立即停止遍历工作。

(4) 执行 take 方法。

经过上述步骤如果查询到了符合条件的出口,则直接执行第 20 行代码,如果没有查询到符合条件的出口,则第 22 行判断 defaultSequenceFlow 是否为空,如果 defaultSequenceFlow 为空则第 30 行代码报错;如果 defaultSequenceFlow 不为空,则第 23 行以该值作为查询条件,检查该值是否是排他网关的出口,如果是则执行 take 方法如第 25 行所示,否则程序报错如第 27 行所示。

14.3 扩展排他网关实战

14.3.1 自定义排他网关行为类

通过分析排他网关行为类的处理逻辑可知排他网关并不能完成上文提出的所有需求,即不能同时执行多条满足条件的出口,这时就需要考虑如何扩展该行为类了,该类的扩展比较灵活,首先自定义一个类,然后继承 ExclusiveGatewayActivityBehavior 类并重写父类中的 leave 方法,具体实现如代码清单 14-4 所示。

代码清单 14-4 ShareniuExclusiveGatewayActivityBehaviorExt.java

```
1 public class ShareniuExclusiveGatewayActivityBehaviorExt extends ExclusiveGatewayActivity-
   Behavior {
2     protected void leave(ActivityExecution execution) {
3         List<PvmTransition> transitionsToTake = new ArrayList<PvmTransition>();
4         // 获取排他网关所有的出线
5         List<PvmTransition> outgoingTransitions = execution.getActivity().getOutgoingTran-
   nsitions();
6         for (PvmTransition outgoingTransition : outgoingTransitions) {
7             // 获取连线中配置的条件表达式
8             Condition condition = (Condition) outgoingTransition.getProperty("condition");
9             // 验证连线中配置的条件表达式是否返回 true
10            boolean evaluate = condition.evaluate(outgoingTransition.getId(), execution);
11            if (evaluate) {
12                transitionsToTake.add(outgoingTransition);
13            }
14        }
```

```
15 execution.takeAll(transitionsToTake, Arrays.asList(execution));
16 }
17 }
```

在上述代码中,第3行声明了存储所有排他网关可以经过的出口集合,第5行获取排他网关所有出口集合,第8行获取出口中配置的条件表达式 condition 对象,第10行验证出口配置的条件决策值是否返回 true,如果是则通过第12行操作将其添加到 transitionsToTake 集合中,第15行直接委托 execution 对象中的 takeAll 方法进行流程实例的运转工作,经过上述一系列步骤之后,排他网关的行为类已经被成功改造了。

14.3.2 自定义活动行为工厂类

接下来需要定义一个类并让其继承 DefaultActivityBehaviorFactory 类,整个过程如代码清单 14-5 所示。

代码清单 14-5 ShareniuActivityBehaviorFactory.java

```
1 public class ShareniuActivityBehaviorFactory extends DefaultActivityBehaviorFactory {
2     // 重写父类中排他网关的分支条件行为类
3     public ExclusiveGatewayActivityBehavior createExclusiveGatewayActivityBehavior(
4         ExclusiveGateway exclusiveGateway) {
5         return new ShareniuExclusiveGatewayActivityBehaviorExt();
6     }
7 }
```

为了实现起来简单,直接将 ShareniuActivityBehaviorFactory 类继承 DefaultActivityBehaviorFactory 类,并重写了父类中的 createExclusiveGatewayActivityBehavior 方法。

强调

ShareniuExclusiveGatewayActivityBehaviorExt 类不能是单例。

14.3.3 替换默认活动行为工厂类

接下来需要做的工作就是将 ShareniuActivityBehaviorFactory 类注入流程引擎配置类,该过程比较简单,直接通过设置 activityBehaviorFactory 开关属性值即可,如代码清单 14-6 所示。

代码清单 14-6 activiti.cfg.xml

```
1 <bean id="processEngineConfiguration"
2     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <!-- 替换默认的活动行为工厂类 -->
4     <property name="activityBehaviorFactory" ref="shareniuActivity" />
5 </bean>
6 <bean id="shareniuActivity" class="com.shareniu.chapter14.ShareniuActivityBehaviorFactory" />
```

接下来部署第 14.2 节定义的流程文档并启动流程实例如代码清单 14-7 所示,进而观察数据库 ACT_RU_TASK 表中的数据变化(如图 14-3 所示),从而验证自定义活动行为类是否已经成功。

代码清单 14-7 App.java

```
1 public void startProcessInstanceById() {  
2     Map<String, Object> map = new HashMap<String, Object>();  
3     map.put("day", 2);  
4     runtimeService.startProcessInstanceById("extensionOperationProcess:1:32504", map);  
5 }
```

ID_	REV_	NAME_	PROC_DEF_ID_
45009	1	项目经理	extensionOperationProcess:1:32504
45011	1	部门主管	extensionOperationProcess:1:32504

图 14-3 启动流程实例之后 ACT_RU_TASK 表数据

由于流程实例启动的时候传递的"day"变量值为 2,而流程文档中定义的 flow8 和 flow3 两条出线均符合条件,所以最终产生的效果如图 14-3 所示,这也侧面验证了自定义排他网关的行为类已经生效,此时此刻是不是觉得很有成就感。

这样的扩展方式虽然完成了上述需求但也衍生了一个新的问题,ShareniuExclusiveGatewayActivityBehaviorExt 类完全覆盖了排他网关中的 leave(ActivityExecution execution)方法。换言之,如果同一个流程文档中定义了多个排他网关,则所有的排他网关都会执行 ShareniuExclusiveGatewayActivityBehaviorExt 类中的 leave(ActivityExecution execution)方法,而不会执行 ExclusiveGatewayActivityBehavior 类中的 leave(ActivityExecution execution)方法,这样就有可能出现问题,因为有些排他网关需要执行自定义的 leave 方法,而有些排他网关需要执行程序默认的 leave 方法,如果存在这样的需求,该怎么办呢?其实也很简单,可以在排他网关中定义一个扩展属性,该属性值决定排他网关行为类是按照扩展的逻辑执行,还是按照默认的方式执行,关于排他网关自定义属性的扩展可以参考第 10.7 节的讲解自行实现,自定义的扩展属性值可以通过 leave(ActivityExecution execution)方法中的 execution 参数进行获取,形如 execution.getActivity().getProperty("shareniu")。

14.4 任务节点处理人多元化配置

在 10.7 节中,讲解了如何自定义任务节点解析类以及如何使用任务监听器的方式将流程文档中配置的任务处理人添加到权限表中,虽然该方式可以实现用户处理人的多元化配置,但是该方式有一个极大的缺陷,就是需要为任务节点配置监听器,只有配置了监听器才可以设置当前节点的处理人,试想一下如果流程文档中有上百个任务节点,难道需要为每一个节点都配置任务监听器吗?显然该方案很不合理,那么能不能使用一种更优雅的方式实现该功能呢?显然可以,这也是接下来需要重点讲解的知识。

14.4.1 任务处理人扩展

首先定义一个流程文档如代码清单 14-8 所示。

代码清单 14-8 usertaskactivitybehavior.bpmn20.xml

```
1 <process id="extensionUserTask" name="ext" isExecutable="true">
2   <startEvent id="start" name="Start"/></startEvent>
3   <userTask id="usertask1" name="usertask1" shareniu:autoUse="true"
4     shareniu:userId="shareniu1,shareniu2"/></userTask>
5   <sequenceFlow id="flow1" sourceRef="start" targetRef="usertask1"/></sequenceFlow>
6   <userTask id="usertask2" name="usertask2"/></userTask>
7   <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"/></sequenceFlow>
8   <endEvent id="endevent1" name="End"/></endEvent>
9   <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="endevent1"/></sequenceFlow>
10 </process>
```

在上述代码中,第 3~4 行为 id 为 usertask1 的任务节点扩展了两个属性,这些扩展属性使用的命名空间为 shareniu,其中 shareniu:autoUse 属性表示当前任务节点是否使用扩展属性为其自动分配处理人,true 表示可以自动分配,shareniu:userId 属性定义了当前任务节点的处理人,可以使用“,”进行分割。

14.4.2 自定义任务解析器

接下来,自定义一个任务解析器用来解析上述的两个属性值,如代码清单 14-9 所示。

代码清单 14-9 ShareniuUserTaskParseHandler.java

```
1 public class ShareniuUserTaskParseHandler extends UserTaskParseHandler {
2   protected void executeParse(BpmnParse bpmnParse, UserTask userTask) {
3     super.executeParse(bpmnParse, userTask);    //调用父类对 userTask 对象进行解析
4     Map<String, List<ExtensionAttribute>> attributes = userTask.getAttributes();
5     //查找 activityImpl 对象
6     ActivityImpl activityImpl = findActivity(bpmnParse, userTask.getId());
7     //将扩展属性值设置到 activityImpl 对象中
8     activityImpl.setProperty("shareniuExt", attributes);
9   }
10 }
```

为了简单起见,ShareniuUserTaskParseHandler 类直接继承 UserTaskParseHandler 并重写父类中的 executeParse 方法,将该方法的处理逻辑总结如下。

(1) 第 3 行调用父类中的 executeParse 方法解析 userTask 对象。

(2) 第 4 行获取自定义的扩展属性值。

(3) 第 6 行通过 findActivity(bpmnParse, userTask.getId()) 方法查找当前任务节点 userTask 在流程虚拟机中的表示对象 activityImpl,因为第 3 行已经成功将 userTask 对象解析转化为 ActivityImpl 实例对象,因此该对象可以直接从流程虚拟机中获取。

(4) 第 8 行将 attributes 集合设置到 activityImpl 对象中, 这样后续流程实例运转时就可以很方便的获取扩展属性值。

14.4.3 自定义任务行为类

自定义一个类继承任务行为类 UserTaskActivityBehavior, 并重写父类中的 handleAssignments 方法, 因为该方法负责为任务节点分配处理人, 相信有了排他网关的学习基础, 实现该功能应该手到擒来, 自定义类如代码清单 14-10 所示。

代码清单 14-10 ShareniuUserTaskActivityBehaviorExt.java

```

1 public class ShareniuUserTaskActivityBehaviorExt extends UserTaskActivityBehavior {
2     public ShareniuUserTaskActivityBehaviorExt(String userTaskId, TaskDefinition t) {
3         super(userTaskId, t);
4     }
5     protected void handleAssignments(Expression assigneeExpression,
6         Expression ownerExpression,
7         Set<Expression> candidateUserExpressions,
8         Set<Expression> candidateGroupExpressions, TaskEntity task,
9         ActivityExecution execution) {
10        PvmActivity activity = execution.getActivity();
11        Object property = activity.getProperty("shareniuExt"); // 获取 shareniuExt 属性值
12        Map<String, List<ExtensionAttribute>> extensionAttribute = null;
13        if (property != null) {
14            extensionAttribute = (Map<String, List<ExtensionAttribute>>) property;
15        }
16        boolean useFlag = false;
17        List<ExtensionAttribute> list = extensionAttribute.get("autoUse"); // 获取 autoUse 属性值
18        for (ExtensionAttribute extensionAttribute2 : list) {
19            String name = extensionAttribute2.getName();
20            String value = extensionAttribute2.getValue();
21            if (Boolean.valueOf(value)) {
22                useFlag = true;
23            }
24        }
25        if (useFlag) {
26            List<ExtensionAttribute> list1 = extensionAttribute.get("userId");
27            for (ExtensionAttribute extensionAttribute2 : list1) {
28                String value = extensionAttribute2.getValue();
29                String[] split = value.split(",");
30                for (String str : split) {
31                    task.addCandidateUser(str); // 为当前节点添加自定义处理人
32                }
33            }
34        }
35        // 调用父类的 handleAssignments 方法分配任务节点的处理人
36        super.handleAssignments(assigneeExpression, ownerExpression,
37            candidateUserExpressions, candidateGroupExpressions, task, execution);
38    }
39 }

```

将 `handleAssignments` 方法的处理逻辑总结如下。

- (1) 第 10 行通过 `execution` 对象获取 `PvmActivity` 实例对象, `PvmActivity` 为 `ActivityImpl` 类的父类。
- (2) 第 11 行通过 `activity` 对象获取 `shareniuExt` 属性值。
- (3) 第 17 行获取 "autoUse" 属性以及该属性值。
- (4) 第 25 行如果 `useFlag` 为 `true`, 则执行第 26~34 行为当前任务节点分配处理人。
- (5) 为了不破坏任务节点处理人的默认处理逻辑, 需要执行第 36~38 行代码。

扩展

在实际项目开发中也可以直接从数据库或者缓存中取出任务节点的扩展处理人并对其进行处理。

当任务的候选人只有一个的时候, 实现自动认领任务功能, 形如 `taskService.claim(taskId, userId)`。

14.4.4 自定义活动行为工厂类

接下来, 要做的事情就是自定义一个类并继承 `DefaultActivityBehaviorFactory` 类, 该实现过程如代码清单 14-11 所示。

代码清单 14-11 `ShareniuActivityBehaviorFactory.java`

```
1 public class ShareniuActivityBehaviorFactory extends DefaultActivityBehaviorFactory {
2     public UserTaskActivityBehavior createUserTaskActivityBehavior(UserTask u, TaskDefinition t) {
3         return new ShareniuUserTaskActivityBehaviorExt(u.getId(), t);
4     }
5 }
```

在上述代码中, 第 3 行直接实例化了 `ShareniuUserTaskActivityBehaviorExt` 类。

接下来需要做的事情就是将 `ShareniuUserTaskParseHandler` 以及 `ShareniuActivityBehaviorFactory` 类注入流程引擎配置类, 整个过程如代码清单 14-12 所示。

代码清单 14-12 `activiti.cfg.xml`

```
6 <bean id="processEngineConfiguration"
7     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
8     <!-- 注入自定义的 ShareniuUserTaskParseHandler 类 -->
9     <property name="customDefaultBpmnParseHandlers">
10         <list>
11             <bean class="com.shareniu.chapter14.ShareniuUserTaskParseHandler"></bean>
12         </list>
13     </property>
14     <!-- 用于替换默认的活动行为工厂类 -->
15     <property name="activityBehaviorFactory" ref="shareniuActivity" />
16 </bean>
17 <bean id="shareniuActivity" class="com.shareniu.chapter14.ShareniuActivityBehaviorFactory" />
```

上述所有工作完毕,部署第 14.4.1 节定义的流程文档,并启动该流程实例,启动流程实例可以参考代码清单 14-7,如不出意外,流程实例启动之后数据库 ACT_RU_IDENTITYLINK 表中数据的变化如图 14-4 所示。

ID	REV	GROUP_ID	TYPE	USER_ID	TASK_ID
65005	1 (Null)		candidate	shareniu1	65004
65006	1 (Null)		participant	shareniu1	(Null)
65007	1 (Null)		candidate	shareniu2	65004
65008	1 (Null)		participant	shareniu2	(Null)

图 14-4 ACT_RU_IDENTITYLINK 表数据变化

14.5 忽略节点使用误区

首先看一下 UserTaskActivityBehavior 类中的 execute 方法执行先后顺序,如代码清单 14-13 所示。

代码清单 14-13 UserTaskActivityBehavior.java

```

1 public void execute(ActivityExecution execution) throws Exception {
2     TaskEntity task = TaskEntity.createAndInsert(execution);
3     ...//省略设置 task 属性
4     handleAssignments(activeAssigneeExpression, activeOwnerExpression,
5         activeCandidateUserExpressions, activeCandidateGroupExpressions, task, execution);
6     if (SkipExpressionUtil.isSkipExpressionEnabled(execution, activeSkipExpression) &&
7         SkipExpressionUtil.shouldSkipFlowElement(execution, activeSkipExpression)) {
8         task.complete(null, false); //完成任务
9     }
10 }

```

在上述代码中,第 4~5 行调用 handleAssignments 方法设置当前任务处理人,第 6~7 行校验当前的节点是否使用了忽略节点功能,如果使用了并且"`_ACTIVITI_SKIP_EXPRESSION_ENABLED`"变量为 true,则第 8 行直接完成该任务。看到这里的处理逻辑,可能会有疑问,第 4~5 行需要将任务节点的处理人添加到 ACT_RU_IDENTITYLINK 表中,ACT_RU_IDENTITYLINK 表中的 TASK_ID 字段对应 ACT_RU_TASK 表中的 ID 字段,如果执行第 8 行代码,则该代码执行完毕之后会将当前节点对应的数据从 ACT_RU_TASK 表中删除,这时再次将处理人的数据添加到 ACT_RU_IDENTITYLINK 表就会出问题,因为 ACT_RU_IDENTITYLINK 表中的 TASK_ID 外键约束缘故。当然如果操作 ACT_RU_TASK 表中的 ASSIGNEE 列,程序不会报错,因为该操作不涉及其他表。

对于任务节点而言,使用忽略节点功能存在如下两个缺陷。

(1) 必须配置表达式 `${_ACTIVITI_SKIP_EXPRESSION_ENABLED}`,其中 `_ACTIVITI_SKIP_EXPRESSION_ENABLED` 必须是 boolean 类型,这样对于开发人员来说配置极不灵活。

(2) 使用忽略节点功能时,如果为任务节点配置了候选人或者候选组,那么当执行上述第 7 行代码时程序报错,报错信息如代码清单 14-14 所示。

代码清单 14-14 报错信息

```
1 com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Cannot add or
update a child row: a foreign key constraint fails ('activiti', 'act_ru_identitylink', CONSTRAINT
'ACT_FK_TSKASS_TASK' FOREIGN KEY ('TASK_ID_') REFERENCES 'act_ru_task' ('ID_'))
```

扩展

Activiti 将数据添加到数据库中分为两个阶段：第一阶段将数据添加到会话缓存；第二阶段将会话缓存中的数据刷新到数据库。

14.6 修复 Activiti 忽略节点 Bug

本案例需在第 14.5 节讲解的案例基础之上进行实现。基于上文的讲解，要想修复 Activiti 忽略节点的 Bug，最通用的方法就是判断当前节点是否使用了忽略节点功能，如果使用了该功能，则不调用 handleAssignments 方法，这个问题就迎刃而解了，具体实现如代码清单 14-15 所示。

代码清单 14-15 ShareniuUserTaskActivityBehaviorExt. java

```
1 protected void handleAssignments(Expression assigneeExpression,
2 Expression ownerExpression,
3 Set<Expression> candidateUserExpressions,
4 Set<Expression> candidateGroupExpressions, TaskEntity task,
5 ActivityExecution execution) {
6     PvmActivity activity = execution.getActivity();
7     Expression skipExpression = task.getTaskDefinition().getSkipExpression();
8     if (!SkipExpressionUtil.isSkipExpressionEnabled(execution, skipExpression)
9         || !SkipExpressionUtil.shouldSkipFlowElement(execution, skipExpression)) {
10         super.handleAssignments(assigneeExpression, ownerExpression,
11             candidateUserExpressions, candidateGroupExpressions, task, execution);
12     }
13 }
```

在上述代码中，第 7 行通过 task 获取 skipExpression 对象，第 8~9 行校验当前节点是否使用了忽略节点功能，如果使用了并且“_ACTIVITI_SKIP_EXPRESSION_ENABLED”变量为 true，就不需要设置任务的候选人或者候选组了，否则第 10~11 行设置任务的候选人或者候选组。

14.7 修复 Activiti 子流程业务键 Bug

在设计流程文档时，通常情况下期望将工作流与项目中的业务进行绑定，因此 Activiti 提供了一个便利的业务键，即 businessKey。但是业务键是流程实例级别的，因此对于需要使用子流程或者引用流程场景下，引擎并不会更新子流程或者引用流程中的业务键，这时就

需要考虑如何修复该 Bug。

首先定义一个流程文档如代码清单 14-16 所示。

代码清单 14-16 subProcess.bpmn

```

1 <process id="subprocess" name="subprocess" isExecutable="true">
2 <startEvent id="startevent1" name="Start"></startEvent>
3 <subProcess id="subprocess1" name="subprocess1">
4 <startEvent id="startevent2" name="Start"></startEvent>
5 <userTask id="usertask1" name="User Task"></userTask>
6 <sequenceFlow id="flow3" sourceRef="startevent2" targetRef="usertask1"></sequenceFlow>
7 <endEvent id="endevent2" name="End"></endEvent>
8 <sequenceFlow id="flow4" sourceRef="usertask1" targetRef="endevent2"></sequenceFlow>
9 </subProcess>
10 <endEvent id="endevent1" name="End"></endEvent>
11 <sequenceFlow id="flow1" sourceRef="subprocess1" targetRef="endevent1"></sequenceFlow>
12 <sequenceFlow id="flow2" sourceRef="startevent1" targetRef="subprocess1"></sequenceFlow>
13 </process>

```

在上述代码中,第 3~9 行定义了 id 为 subprocess1 的子流程。将该流程文档进行部署并启动流程实例,在启动流程实例的同时设置业务键为 sharenui,如代码清单 14-17 所示。

代码清单 14-17 App.java

```

1 public void startProcessInstanceId() throws IOException {
2     runtimeService.startProcessInstanceId("subprocess:1:4", "sharenui");
3 }

```

上述操作完毕,查看 ACT_RU_EXECUTION 表中的数据变化如图 14-5 所示。

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_
2501	1	2501	sharenui	(Null)	subprocess:1:4
2503	1	2501	(Null)	2501	subprocess:1:4

图 14-5 ACT_RU_EXECUTION 表数据变化

在图 14-5 中,ID_为 2503 的数据 BUSINESS_KEY_列的值为空,Activiti 确实没有更新子流程中的业务键。

上述的流程文档中,id 为 startevent1 的开始节点执行完毕,引擎会根据子流程的行为类决定流程实例该如何运转,既然子流程的行为类没有更新业务键,那么接下来的工作重点就是修改子流程行为类中的代码,子流程的行为类为 SubProcessActivityBehavior。

首先,定义一个类并继承 SubProcessActivityBehavior 类,如代码清单 14-18 所示。

代码清单 14-18 SharenuiSubProcessActivityBehavior.java

```

1 public class SharenuiSubProcessActivityBehavior extends SubProcessActivityBehavior{
2     public void execute(ActivityExecution execution) throws Exception{
3         ActivityExecution parent = execution.getParent(); //获取流程实例
4         ExecutionEntity entity = null;

```

```

5      if(null!= parent){
6          String businessKey = parent.getProcessBusinessKey(); //获取业务键
7          entity = (ExecutionEntity) execution;
8          entity.setBusinessKey(businessKey); //设置子流程的业务键
9      }
10     super.execute(entity); //调用父类中的 execute 方法
11 }
12 }

```

上面代码中已经看到了 DataSource 初始化的雏形,其执行逻辑总结如下。

在上述代码中,第3行获取 ActivityExecution 实例对象,第6行获取流程实例中的业务键,第8行将业务键设置到 entity 对象中,第10行调用 SubProcessActivityBehavior 类中的 execute 方法。

接着自定义一个类并继承 DefaultActivityBehaviorFactory 类,该实现过程如代码清单 14-19 所示。

代码清单 14-19 ShareniuActivityBehaviorFactory.java

```

1 public class ShareniuActivityBehaviorFactory extends DefaultActivityBehaviorFactory {
2     public SubProcessActivityBehavior createSubprocActivityBehavior(SubProcess subProcess) {
3         return new ShareniuSubProcessActivityBehavior();
4     }
5 }

```

第3行实例化 ShareniuSubProcessActivityBehavior 类。接下来需要将 ShareniuActivityBehaviorFactory 注入流程引擎配置类,可以参考第 14.3.3 节的讲解。

最后再次执行代码清单 14-17 进而观察 ACT_RU_EXECUTION 表中的数据变化如图 14-6 所示。

ID	REV	PROC_INST_ID	BUSINESS_KEY	PARENT_ID	PROC_DEF_ID
2501	1	2501	shareniu	(Null)	subprocess:1:4
2503	1	2501	(Null)	2501	subprocess:1:4
5001	1	5001	shareniu	(Null)	subprocess:1:4
5003	1	5001	shareniu	5001	subprocess:1:4

图 14-6 ACT_RU_EXECUTION 表数据变化

通过图 14-6 可知,子流程中的业务键已经被成功更新。

扩展

引用流程的行为类为 CallActivityBehavior。

第 15 章

Activiti 存储之 MyBatis

MyBatis 是一个轻量级的数据持久层 ORM 映射框架, 每一个使用 MyBatis 操作数据库的工程都是基于 SqlSessionFactory 实例对象实现的, 该实例负责对外提供数据库的连接和操作。Activiti 使用 MyBatis 作为数据访问层, 正因为如此, 本章重点讲解 Activiti 是如何封装 MyBatis 的。

15.1 初始化 dataSource

首先思考一个问题, dataSource 数据源是如何被初始化的, dataSource 的初始化是从 ProcessEngineConfigurationImpl 类的 initDataSource 方法开始的, 以该方法为切入点深入探究 dataSource 的初始化过程, 该方法执行逻辑如代码清单 15-1 所示。

代码清单 15-1 ProcessEngineConfigurationImpl.java

```
1 protected void initDataSource() {
2     if (dataSource == null) {
3         if (dataSourceJndiName != null) {
4             try {
5                 dataSource = (DataSource) new InitialContext().lookup(dataSourceJndiName);
6             } catch (Exception e) {
7                 throw new ActivitiException("couldn't lookup datasource from ");
8             }
9         } else if (jdbcUrl != null) {
10            if ( ( jdbcDriver == null ) || ( jdbcUrl == null ) || ( jdbcUsername == null ) ) {
11                throw new ActivitiException("DataSource or JDBC properties have to be ");
12            }
13            PooledDataSource pooledDataSource =
14                new PooledDataSource(ReflectUtil.getClassLoader(), jdbcDriver, jdbcUrl,
15                jdbcUsername, jdbcPassword );
```

```

16      ...//省略填充 pooledDataSource 对象的属性值
17    }
18    if (databaseType == null) {
19        initDatabaseType();
20    }
21 }

```

从上面代码中已经看到了 dataSource 初始化的雏形,其执行逻辑总结如下。

(1) 第 2 行判断 dataSource 开关属性值是否为空。

(2) 第 3 行判断 dataSourceJndiName 开关属性值是否为空,如果不为空,则第 5 行开始查找并尝试将其转化为 dataSource 对象,JNDI 方式配置数据源的过程可以搜索相关资料自行学习。

(3) 如果以上两个开关属性值经过判断之后都为空,则第 9 行判断 jdbcUrl 开关属性值是否为空,如果不为空,则说明开发人员已经配置了该属性值,首先第 10 行对 jdbcDriver 开关属性值和 jdbcUsername 开关属性值进行非空校验,如果这两个属性值任意一个为空,则第 11 行程序报错,否则执行如下逻辑。

(4) 第 13~15 行实例化 PooledDataSource 类,并为 pooledDataSource 对象填充属性值。

(5) 上述一系列操作执行完毕,第 18 行开始判断 databaseType 开关属性值是否为空,如果为空,则第 19 行开始执行 initDatabaseType 方法,该方法的具体实现如代码清单 15-2 所示。

代码清单 15-2 ProcessEngineConfigurationImpl.java

```

1 public void initDatabaseType() {
2     Connection connection = null;
3     try {
4         connection = dataSource.getConnection(); //打开连接
5         //获取数据库生产厂商信息
6         DatabaseMetaData databaseMetaData = connection.getMetaData();
7         //例如 MySQL 数据库的 databaseProductName 值为 MySQL
8         String databaseProductName = databaseMetaData.getDatabaseProductName();
9         //从集合中获取数据库类型
10        databaseType = databaseTypeMappings.getProperty(databaseProductName);
11        if (databaseType == null) {
12            throw new ActivitiException("couldn't deduct database type from database");
13        }
14    } catch (SQLException e) {
15        log.error("Exception while initializing Database connection", e);
16    } finally {
17        try {
18            if (connection != null) {
19                connection.close(); //关闭连接
20            }
21        } catch (SQLException e) {
22            log.error("Exception while closing the Database connection", e);

```



```

23     }
24 }
25 }

```

虽然上面方法的处理步骤看似简单,但是每个步骤都非常重要,可以自行结合代码和部分注释进行理解。这里为何需要打开一次数据库连接获取数据库类型呢?暂且留下一个悬念,稍后详细讲解。下面讲解 10 行 databaseTypeMappings 集合,相关实现如代码清单 15-3 所示。

代码清单 15-3 ProcessEngineConfigurationImpl.java

```

1  protected static Properties databaseTypeMappings = getDefaultDatabaseTypeMappings();
2  public static final String DATABASE_TYPE_DB2 = "db2";
3  protected static Properties getDefaultDatabaseTypeMappings() {
4      Properties databaseTypeMappings = new Properties();
5      databaseTypeMappings.setProperty("H2", DATABASE_TYPE_H2);
6      ...
7      return databaseTypeMappings;
8  }

```

getDefaultDatabaseTypeMapping 方法内部使用 Properties 类封装了所有的数据库厂商信息,从而解决同一个数据库存在多个版本的问题,进而把同种类型但版本不同的数据库进行统一对待。例如 DB2 数据库提供了很多发行版本,不管客户端使用哪个版本,经过这里的处理之后,统一按照 db2 类型进行处理。

获取数据库类型的目的是为了下一步对数据库的 SQL 语句进行差异化处理做铺垫的,因此获取数据库类型的操作是非常重要的。通过分析上文代码的处理逻辑可以看出只有 databaseType 开关属性值为空,程序才会执行 initDatabaseType 方法中的逻辑,而 initDatabaseType 方法的处理就需要打开一次数据库连接进行数据库类型的获取,仅仅是为了获取数据库的类型就打开一次数据库连接,有点得不偿失,所以可以直接在流程引擎配置类中配置 databaseType 开关属性值。

注意

可以直接在流程引擎配置类中设置 databaseType 开关属性值为具体的数据库类型,避免程序仅仅为了获取数据库的类型就打开一次数据库连接,这也是性能优化的地方之一。

15.2 Activiti 数据访问层关系分析

在使用 MyBatis 时,首先要做的工作就是配置一系列的映射文件,从而使实体类与表一一映射。MyBatis 数据库表的操作都是基于 SqlSessionFactory 实例对象进行的,SqlSessionFactory 的默认实现类为 DefaultSqlSessionFactory,因此要想使用 MyBatis,获取 SqlSessionFactory 实例是一个必不可少的环节。

15.2.1 实体类与数据库表的映射

接下来,分析 Activiti 是如何封装数据访问层的,Activiti 与 MyBatis 框架集成时,会读取项目中配置的 MyBatis 映射文件,该文件内容如代码清单 15-4 所示。

代码清单 15-4 org/activiti/db/mapping/mappings.xml

```
1 <?xml version = "1.0" encoding = "UTF - 8"?>
2 <configuration>
3   <settings><setting name = "lazyLoadingEnabled" value = "false" /></settings>
4   <typeAliases>
5     <typeAlias type = "org.activiti.engine.impl.persistence.ByteArrayRefTypeHandler"
alias = "ByteArrayRefTypeHandler"/>
6   </typeAliases>
7   <typeHandlers>
8     <typeHandler handler = "ByteArrayRefTypeHandler"
9       javaType = "org.activiti.engine.impl.persistence.entity.ByteArrayRef"
10      jdbcType = "VARCHAR"/>
11   </typeHandlers>
12   <mappers>
13     <mapper resource = "org/activiti/db/mapping/entity/Attachment.xml" />
14     ...//省略映射文件
15   </mappers>
16 </configuration>
```

通过上面 XML 文件的内容可以看出,第 4~11 行使用了类型转换器 ByteArrayRefTypeHandler,该类型转换器所做的工作就是将数据库中的 VARCHAR 类型与 Java 中的 ByteArrayRef 类型进行相互转换。

第 3 行 settings 节点的子节点 setting 的 lazyLoadingEnabled 属性值为 false,表示禁用延迟加载。

第 12~15 行 mappers 节点中注册了所有 Activiti 操作数据表所需要的映射文件,接下来看一下映射文件、数据库表、实体类三者之间的关系如表 15-1 所示。

表 15-1 映射文件、数据库表、实体类之间的关系

序号	映射文件名	数据库表名	实体类名
1	Attachment.xml	ACT_HI_ATTACHMENT	AttachmentEntity
2	ByteArray.xml	ACT_GE_BYTEARRAY	ByteArrayEntity
3	Comment.xml	ACT_HI_COMMENT	CommentEntity
4	Deployment.xml	ACT_RE_DEPLOYMENT	DeploymentEntity
5	Execution.xml	ACT_RU_EXECUTION	ExecutionEntity
6	Group.xml	ACT_ID_GROUP	GroupEntity
7	HistoricActivityInstance.xml	ACT_HI_ACTINST	HistoricActivityInstanceEntity
8	HistoricDetail.xml	ACT_HI_DETAIL	HistoricFormPropertyEntity
9	HistoricProcessInstance.xml	ACT_HI_PROCINST	HistoricProcessInstanceEntity
10	HistoricVariableInstance.xml	ACT_HI_VARINST	HistoricVariableInstanceEntity

续表

序号	映射文件名	数据库表名	实体类名
11	HistoricTaskInstance.xml	ACT_HI_TASKINST	HistoricTaskInstanceEntity
12	HistoricIdentityLink.xml	ACT_HI_IDENTITYLINK	HistoricIdentityLinkEntity
13	IdentityInfo.xml	ACT_ID_INFO	IdentityInfoEntity
14	IdentityLink.xml	ACT_RU_IDENTITYLINK	IdentityLinkEntity
15	Job.xml	ACT_RU_JOB	JobEntity
16	Membership.xml	ACT_ID_MEMBERSHIP	MembershipEntity
17	Model.xml	ACT_RE_MODEL	ModelEntity
18	ProcessDefinition.xml	ACT_RE_PROCDEF	ProcessDefinitionEntity
19	ProcessDefinitionInfo.xml	ACT_PROCDEF_INFO	ProcessDefinitionInfoEntity
20	Property.xml	ACT_GE_PROPERTY	PropertyEntity
21	Resource.xml	ACT_GE_BYTEARRAY	ResourceEntity
22	TableData.xml	引擎所在数据库的所有表	无
23	Task.xml	ACT_RU_TASK	TaskEntity
24	User.xml	ACT_ID_USER	UserEntity
25	VariableInstance.xml	ACT_RU_VARIABLE	VariableInstanceEntity
26	EventSubscription.xml	ACT_RU_EVENT_SUBSCR	EventSubscriptionEntity
27	EventLogEntry.xml	ACT_EVT_LOG	EventLogEntryEntity

注意

以上实体类所在的路径为 org.activiti.engine.impl.persistence.entity, 建议结合映射文件内容逐个查看。

15.2.2 实例化 SqlSessionFactory

下面详细分析 Activiti 是如何处理 SqlSessionFactory 类的, 首先看一下 SqlSessionFactory 类的初始化过程, 如代码清单 15-5 所示。

代码清单 15-5 ProcessEngineConfigurationImpl.java

```
1 final String DEFAULT_MYBATIS_MAPPING_FILE = "org/activiti/db/mapping/mappings.xml";
2 protected void initSqlSessionFactory() {
3     if (sqlSessionFactory == null) {
4         InputStream inputStream = null;
5         try {
6             inputStream = getMyBatisXmlConfigurationStream();
7             Environment environment = new Environment("default", transactionFactory,
8             dataSource);
9             Reader reader = new InputStreamReader(inputStream);
10            Properties properties = new Properties();
11            properties.put("prefix", databaseTablePrefix);
12            if(databaseType != null) {
13                ...//省略设置值
14            }
15            Configuration configuration = initMybatisConfiguration(environment, reader,
```

```

16 properties);
17 sqlSessionSessionFactory = new DefaultSqlSessionFactory(configuration);
18 } catch (Exception e) {
19 throw new ActivitiException("Error while building ibatis SqlSessionFactory");
20 } finally {
21 IoUtil.closeSilently(inputStream);
22 }
23 }
24 }

```

以上代码中大部分操作都是基于 MyBatis 框架提供的 API, 如果接触 MyBatis 不多的话, 可能不容易理解, 但是没关系, 上面方法的处理逻辑也很简单, 都是 MyBatis 的基本用法, 将其处理流程总结如下。

(1) 第 3 行判断 sqlSessionSessionFactory 开关属性值是否为空, 如果不为空, 那么 Activiti 就无须关心该类的实例化工作, 因此也就不需要实现; 否则开始执行后续的处理逻辑。

(2) 第 6 行调用 getMyBatisXmlConfigurationStream 方法获取 Mapper(映射文件)的数据流, 该 Mapper 文件所在的位置如第 1 行定义所示, 如果打算自定义 Mapper 文件, 则可以自定义一个类, 然后继承当前类(流程引擎配置类)并重写该方法即可。

(3) 第 7~8 行实例化 Environment 类, 并将 transactionFactory 和 dataSource 作为该类中构造方法的参数进行传递。

(4) 第 9 行实例化文件流读取器 Reader 类。

(5) 第 10~14 实例化 Properties 类并根据数据库的类型设置不同的属性值, 常用的属性如下: 数据库表的前缀 prefix、limitBefore、limitAfter、limitBetween、limitOuterJoinBetween、orderBy、limitBeforeNativeQuery, 根据属性名称也可以看出这些属性值和 SQL 分页以及排序有关, 为何需要该操作? 该操作的意义何在? 因为不同的数据库 SQL 语法有一定差异, 这些属性值就是为了解决这个问题, 关于属性值的获取工作暂且留下悬念, 稍后加以说明。

(6) 第 15 行实例化 Configuration 类, MyBatis 中所有的信息都存储在该类中, 可以将该类理解为 MyBatis 的注册中心, 该类提供了一系列设置属性的方法, 可以通过读取配置文件, 也可以通过程序进行设置。initMybatisConfiguration 方法的具体实现如代码清单 15-6 所示。

代码清单 15-6 ProcessEngineConfigurationImpl.java

```

1 protected Configuration initMybatisConfiguration(Environment environment, Reader reader,
Properties properties) {
2     XMLConfigBuilder parser = new XMLConfigBuilder(reader, "", properties);
3     Configuration configuration = parser.getConfiguration();
4     configuration.setEnvironment(environment);
5     initMybatisTypeHandlers(configuration);
6     initCustomMybatisMappers(configuration);
7     configuration = parseMybatisConfiguration(configuration, parser);
8     return configuration;
9 }

```


根据上面的代码可知,initMybatisConfiguration 方法负责整个配置类 Configuration 的实例化以及属性填充工作,将该方法的执行步骤以及实现功能总结如下。

- 第 2 行实例化 XMLConfigBuilder 类,该类主要用于构造 Configuration 实例对象。
- 第 5 行 initMybatisTypeHandlers 方法用于注册类型转换器,该方法的具体实现如代码清单 15-7 所示。

代码清单 15-7 ProcessEngineConfigurationImpl.java

```
1 protected void initMybatisTypeHandlers(Configuration configuration) {
2     configuration.getTypeHandlerRegistry().register(VariableType.class, JdbcType.VARCHAR, new
3     IbatisVariableTypeHandler());
4 }
```

通过以上代码可以看出,initMybatisTypeHandlers 方法负责向 MyBatis 注册 IbatisVariableTypeHandler 类,该类可以实现数据库中的 VARCHAR 类型与 Activiti 中的 VariableType 类型相互转换,该操作非常重要,Activiti 中可以使用的变量类型均通过 IbatisVariableTypeHandler 类进行转换的。

- initCustomMybatisMappers 方法用于注册 Mapper 接口(映射类)如代码清单 15-8 所示。

代码清单 15-8 ProcessEngineConfigurationImpl.java

```
1 protected Set<Class<?>> customMybatisMappers; //开关属性
2 public Set<Class<?>> getCustomMybatisMappers() { //获取 customMybatisMappers 值
3     return customMybatisMappers;
4 }
5 protected void initCustomMybatisMappers(Configuration configuration) {
6     if (getCustomMybatisMappers() != null) { //判断 customMybatisMappers 值是否为空
7         for (Class<?> clazz : getCustomMybatisMappers()) { //循环遍历 customMybatisMappers
8             configuration.addMapper(clazz); //注册到 configuration 对象中
9         }
10    }
```

如果开发人员打算使用自定义映射类(注解方式),可以直接设置 customMybatisMappers 开关属性值。

- parseMybatisConfiguration 方法用于注册 Mapper 映射文件(XML 配置)如代码清单 15-9 所示。

代码清单 15-9 ProcessEngineConfigurationImpl.java

```
1 protected Set<String> customMybatisXMLMappers; //自定义 MyBatis 中的 XML 映射文件
2 public Set<String> getCustomMybatisXMLMappers() { //获取 customMybatisXMLMappers 值
3     return customMybatisXMLMappers;
4 }
5 protected Configuration parseMybatisConfiguration(Configuration configuration, XMLConfigBuilder
6     parser) {
7     return parseCustomMybatisXMLMappers(parser.parse());
```

```

7 }
8 protected Configuration parseCustomMybatisXMLMappers(Configuration configuration) {
9     if (getCustomMybatisXMLMappers() != null) //判断 customMybatisXMLMappers 集合是否为空
10     for(String resource: getCustomMybatisXMLMappers()){ //循环遍历 customMybatisXMLMappers
11         XMLMapperBuilder mapperParser = new XMLMapperBuilder(getResourceAsStream(resource),
12             configuration, resource, configuration.getSqlFragments());
13         mapperParser.parse(); //开始解析
14     }
15     return configuration;
16 }

```

parseMybatisConfiguration 方法直接委托第 8 行定义的 parseCustomMybatisXMLMappers 方法处理 XML 映射文件, parseCustomMybatisXMLMappers 方法首先判断 customMybatisXMLMappers 集合是否为空, 如果不为空, 第 10~14 就开始遍历该集合并获取元素值, 最终将其注册到 Configuration 实例对象中。

(7) 第 17 行实例化 DefaultSqlSessionFactory 类。

15.3 自定义 Mapper 实战

在实际项目开发过程中, 如果觉得 Activiti 中一些默认的 SQL 语句不灵活, 不能完全符合自己的业务场景, 可以定制更加符合自己业务场景的 SQL 语句并通过自定义 Mapper 的方式执行自定义 SQL 语句。下面讲解如何自定义 Mapper 并将其注入流程引擎配置类。

15.3.1 自定义 Mapper

有这样一个场景, 需要查询 ACT_RU_TASK 表中的数据, 但是只希望查询 ID_ 和 NAME_ 两个列, 自定义类(基于注解方式实现)的实现如代码清单 15-10 所示。

代码清单 15-10 ShareniuMapper.java

```

1 public interface ShareniuMapper {
2     @Select("SELECT ID_ as id, NAME_ as name FROM ACT_RU_TASK")
3     List<Map<String, Object>> selectTasks();
4 }

```

ShareniuMapper 接口定义了 selectTasks 方法用于查询 ACT_RU_TASK 表中的 ID_ 和 NAME_ 两列的值, 接下来的工作就是将 ShareniuMapper 注入流程引擎配置类, 如代码清单 15-11 所示。

代码清单 15-11 activiti.cfg.xml

```

1 <bean id="processEngineConfiguration"
2     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <property name="customMybatisMappers">

```

```

4      <set><!-- 设置 customMybatisMappers 属性值 -->
5      <value>com.shareniu.chapter15.ShareniuMapper</value>
6    </set>
7  </property>
8 </bean>

```

在上述代码中,第3行通过设置 customMybatisMappers 开关属性即可完成自定义 Mapper 注入流程引擎配置类。

以上工作完成之后,如何调用自定义 SQL? Activiti 为开发人员提供了 ManagementService 服务类,该类在 Activiti5.15 版本中提供了一系列直接与数据库打交道的方法,其中就包括执行 SQL 语句的方法 executeCustomSql,该方法允许程序执行 CustomSqlExecution 实例对象,相关实现如代码清单 15-12 所示。

代码清单 15-12 App.java

```

1 public void testCustomSqlExecution() {
2     ManagementService managementService = processEngine.getManagementService();
3     // 实例化 CustomSqlExecution 类,这里使用匿名内部类
4     CustomSqlExecution<ShareniuMapper, List<Map<String, Object>>> customSqlExecution = new
5     AbstractCustomSqlExecution<ShareniuMapper, List<Map<String, Object>>>(ShareniuMapper.
6     class)
7     {
8         public List<Map<String, Object>> execute(ShareniuMapper customMapper) {
9             // 省略查询数据库操作
10             return customMapper.selectTasks();
11         }
12     };
13     // 执行 executeCustomSql 方法
14     List<Map<String, Object>> results = managementService.executeCustomSql(custom-
15     SqlExecution);
16     System.out.println(results); // 输出从数据库中查询到的值
17 }

```

执行上面的代码,如不出意外应该可以看到期望输出的结果。本案例查询的是 ACT_RU_TASK 表中的数据,当然,自定义 Mapper 类可以查询流程引擎所连接的数据库中的任意表。

15.3.2 自定义 SQL 执行原理

上文使用了 ManagementService 类中的 executeCustomSql 方法执行自定义 Mapper 映射类中的查询方法,在 executeCustomSql 方法中需要构造 CustomSqlExecution 实例对象,接下来分析 executeCustomSql 方法的内部实现原理,首先查看 CustomSqlExecution 类的功能结构图如图 15-1 所示。

(1) CustomSqlExecution: 该接口定义了获取 Mapper 映射类的方法 getMapperClass 和执行 Mapper 类的方法 execute。

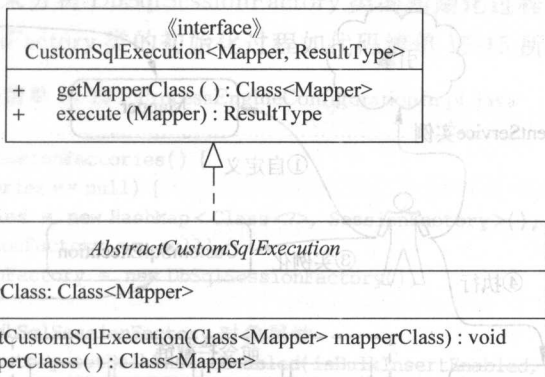


图 15-1 CustomSqlExecution 类的功能结构图

(2) AbstractCustomSqlExecution: 该抽象类实现了 CustomSqlExecution 接口, 并对 getMapperClass 方法提供了实现。由于该抽象类中没有显式定义无参构造方法, 因此开发人员自定义类并继承该类时必须有一个有参的构造方法(参数的个数以及类型需要与父类中的有参构造方法一致)。

了解了 CustomSqlExecution 类的设计原理之后接着分析 ManagementServiceImpl 类中的 executeCustomSql 方法, 如代码清单 15-13 所示。

代码清单 15-13 ManagementServiceImpl.java

```
1 public < MapperType, ResultType > ResultType executeCustomSql (CustomSqlExecution < MapperType,
ResultType> customSqlExecution) {
2     Class< MapperType > mapperClass = customSqlExecution.getMapperClass();
3     return commandExecutor.execute(new ExecuteCustomSqlCmd < MapperType, ResultType >(mapperClass,
4     customSqlExecution));
5 }
```

executeCustomSql 方法仅仅是通过 customSqlExecution 参数值获取到 Class< MapperType > 实例对象, 然后直接使用命令执行器 commandExecutor 执行 ExecuteCustomSqlCmd 命令类, 该命令类的核心定义如代码清单 15-14 所示。

代码清单 15-14 ExecuteCustomSqlCmd.java

```
1 public ResultType execute(CommandContext commandContext) {
2     Mapper mapper = commandContext.getDbSqlSession().getSqlSession().getMapper(mapperClass);
3     return customSqlExecution.execute(mapper);
4 }
```

execute 方法首先通过 commandContext 对象获取 DbSqlSession 实例对象, 然后通过 DbSqlSession 实例对象获取 SqlSession 实例对象, SqlSession 类正是 MyBatis 中的核心类, 该类在平时开发中使用较多, 因此比较熟悉。DbSqlSession 实例对象在哪里进行初始化, 又是负责完成什么工作, 关于这一点, 暂且留下悬念, 稍后详细说明。使用图 15-2 对自定义 Mapper 映射类的执行过程进行通俗易懂的描述。

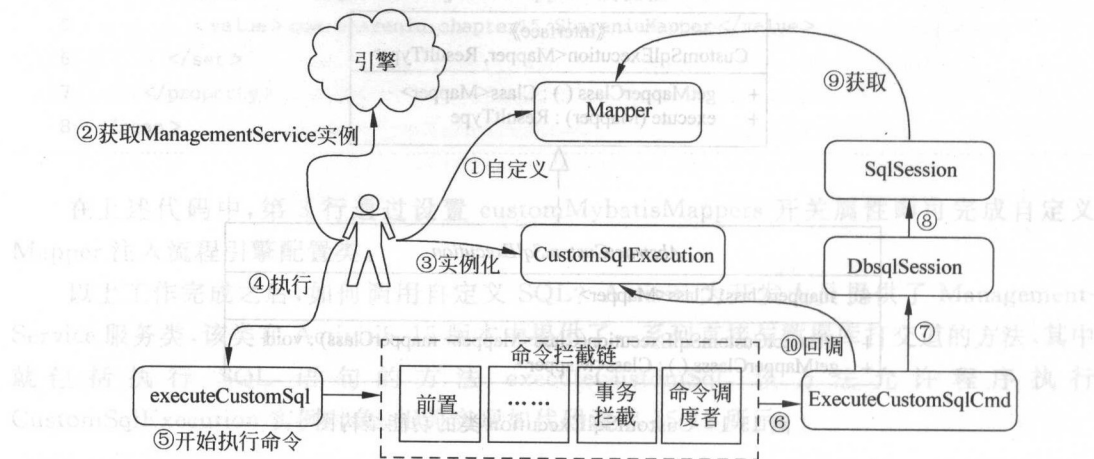


图 15-2 自定义 Mapper 执行过程

15.4 SessionFactory

15.4.1 初始化 SessionFactory

上文着重讲解了 SqlSessionFactory 类的实例化过程,该类主要用于对外提供数据库的连接以及数据库的访问操作,但是 Activiti 并没有直接对外部程序提供 SqlSessionFactory,而是使用 DbSqlSessionFactory 类包裹了 SqlSessionFactory 类,如图 15-3 所示。

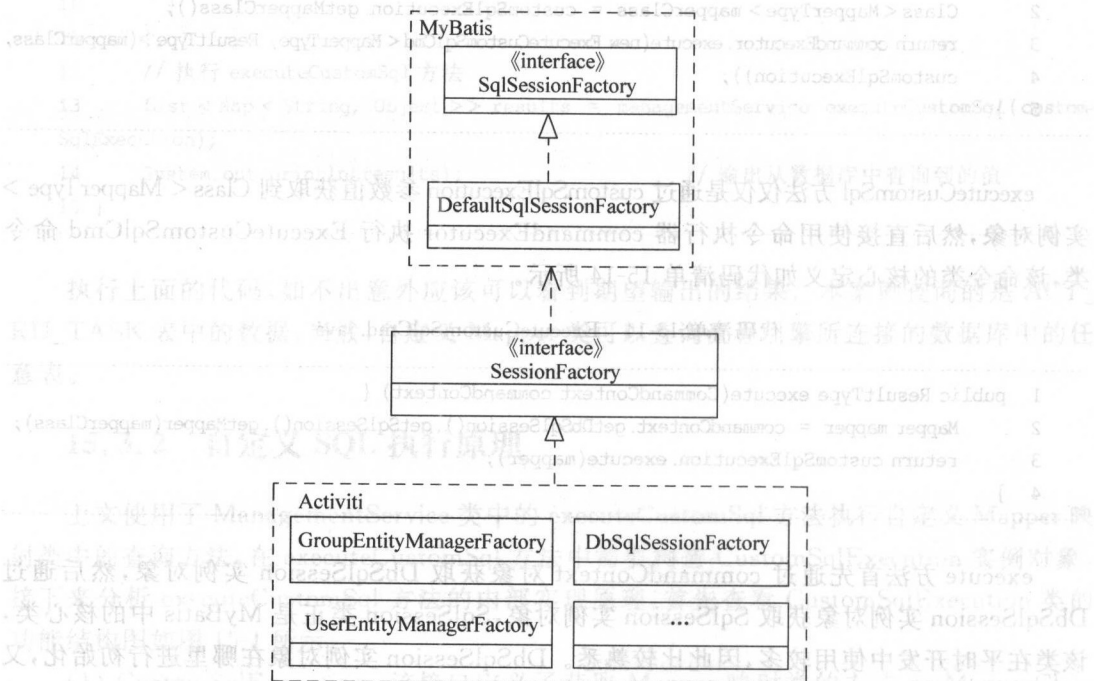


图 15-3 SqlSessionFactory 与 DbSqlSessionFactory 的关系

真的是这样吗? 接下来分析 DbSqlSessionFactory 类的初始化过程, 进而验证以上所说的是否正确, DbSqlSessionFactory 类的初始化过程如代码清单 15-15 所示。

代码清单 15-15 ProcessEngineConfigurationImpl.java

```

1  protected void initSessionFactories() {
2      if (sessionFactories == null) {
3          sessionFactories = new HashMap<Class<?>, SessionFactory>();
4          if (dbSqlSessionFactory == null) {
5              dbSqlSessionFactory = new DbSqlSessionFactory();
6          }
7          ....//省略设置 dbSqlSessionFactory 对象属性
8          dbSqlSessionFactory.setBulkInsertEnabled(isBulkInsertEnabled, databaseType);
9          addSessionFactory(dbSqlSessionFactory);
10         addSessionFactory(new GenericManagerFactory(AttachmentEntityManager.class));
11         ....//省略一系列实体管理类的添加
12         addSessionFactory(new MembershipEntityManagerFactory());
13     }
14     if (customSessionFactories != null) {
15         for (SessionFactory sessionFactory: customSessionFactories) {
16             addSessionFactory(sessionFactory);
17         }
18     }
19 }

```

initSessionFactories 方法的处理逻辑比较复杂, 将其梳理总结如下。

(1) 第 2 行判断 sessionFactories 开关属性值是否为空, 如果为空, 则开始执行如下逻辑, 否则直接执行第 14~18 行代码。

(2) 第 4 行判断 dbSqlSessionFactory 开关属性值是否为空, 如果为空则第 5 行直接实例化 DbSqlSessionFactory 类。

(3) 第 7~8 行对 dbSqlSessionFactory 对象进行属性填充, 该对象需要填充的属性值包括数据库表前缀是否支持批量操作、批量操作的限制、是否使用权限表(用户组织架构)、是否使用历史表等。

(4) 第 9~12 行调用 addSessionFactory 方法将各种实体管理类添加到 sessionFactories 集合中, addSessionFactory 方法的详细实现如代码清单 15-16 所示。

代码清单 15-16 ProcessEngineConfigurationImpl.java

```

1  protected Map<Class<?>, SessionFactory> sessionFactories;
2  protected void addSessionFactory(SessionFactory sessionFactory) {
3      sessionFactories.put(sessionFactory.getSessionType(), sessionFactory);
4  }

```

该方法主要是将实体管理类添加到 sessionFactories 集合中, 由于篇幅有限, 此代码清单中对于该过程的代码省略了很多, 可以跟进该代码自行学习。

sessionFactories 集合为 Map 数据结构, 其中 key 为 sessionFactory 对象的 getSessionType 方法的返回值, value 为 sessionFactory 对象。看到 Map 数据结构可知相同的 key 值只能存在一个, 因此开发人员自定义的 SessionFactory 类可以替换系统内置的 SessionFactory 类。

(5) 第 14 行判断 customSessionFactories 开关属性值是否为空,如果不为空则调用 addSessionFactory 方法进行操作。

15.4.2 SessionFactory 架构

上面提到一系列的实体管理类通过 addSessionFactory 方法注册到 sessionFactories 集合中,接下来详细分析 SessionFactory,首先讲解该接口的功能架构,如图 15-4 所示。

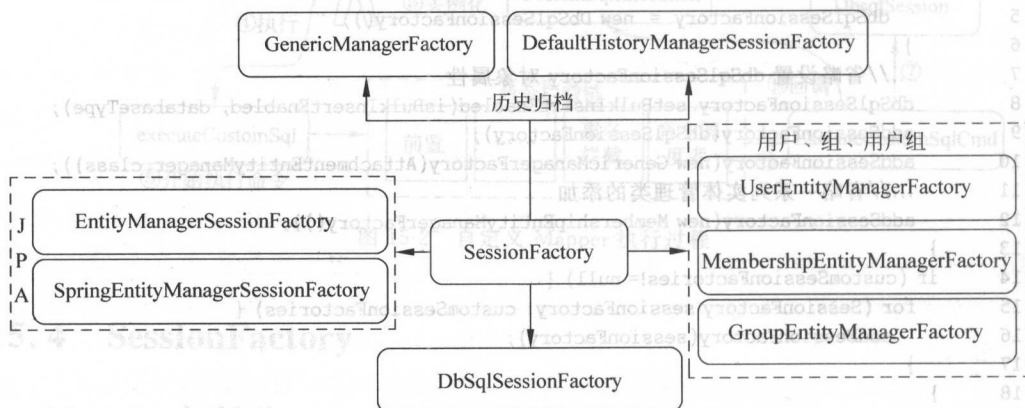


图 15-4 SessionFactory 功能架构图

上图根据 SessionFactory 接口的实现类的功能职责进行了划分,详细说明如下。

(1) SessionFactory: 该接口定义了 openSession 方法(负责连接和操作数据库)和 getSessionType 方法(用于返回实体管理类)。

(2) DefaultHistoryManagerSessionFactory: 该类负责维护 DefaultHistoryManager 历史实体管理类,为什么需要将历史实体管理类进行单独维护呢?第 11.3.4 节讲解了 HistoryLevel 枚举类,该类定义了不同的历史归档级别,DefaultHistoryManager 类负责与数据库中的历史表打交道,该类定义了一系列操作历史表的方法,每一个方法的处理步骤均是首先获取流程引擎配置类中的 historyLevel 值(开关属性),然后根据该值的级别决定历史数据是否可以入库、删除、更新操作,通过该类的职责可知,如果开发人员觉得 Activiti 对历史数据处理不够友好,可以直接自定义一个类并重写 DefaultHistoryManager 类中的相应方法,然后将自定义类交给 DefaultHistoryManagerSessionFactory 类进行管理,从而可以手动控制历史数据。

(3) EntityManagerSessionFactory 和 SpringEntityManagerSessionFactory 类负责管理 JPA 中的实体管理类。

(4) UserEntityManagerFactory: 该类负责管理 UserEntityManager 类(管理 ACT_ID_USER 表或者视图)。

(5) GroupEntityManagerFactory: 该类负责管理 GroupEntityManager 类(管理 ACT_ID_GROUP 表或者视图)。

(6) MembershipEntityManagerFactory: 该类负责管理 MembershipEntityManager 类(管理 ACT_ID_MEMBERSHIP 表或者视图)。

以上所说的 UserEntityManagerFactory、GroupEntityManagerFactory、Membership-

EntityManagerFactory 三个类分别用于处理用户、组、用户组关系表,这里进行单独定义是为了方便客户端灵活修改和扩展,例如开发人员完全可以通过扩展以上几个类对用户、组、用户组关系表进行自定义实现。

(7) DbSessionFactory: 该类非常重要,定义了一系列获取 SQL 执行 id 值的方法(如 <insert id="bulkInsertMembership",则这里的 SQL 执行 id 值为 bulkInsertMembership)。该类中的静态代码块非常重要,负责为不同类型的数据库量身定制不同的 SQL 执行 id 值。

(8) GenericManagerFactory: 通用实体管理工厂类,该类负责维护以上所说之外的实体管理类,这些管理类均需要实现 Session 接口。

15.5 Session

上文重点讲解了 SqlSessionFactory 和 DbSessionFactory 的初始化过程以及两者之间的关系,SqlSession 对应一次数据库会话操作,创建 SqlSession 实例入口只有一个,只能通过 SqlSessionFactory 的 openSession 方法进行创建。

因为程序与数据库的会话是短暂的,所以 SqlSession 的生命周期与会话绑定在一起,每次访问数据库时都需要创建 SqlSession 实例,会话关闭,如果需要再次与数据库会话,就需要重新创建 SqlSession 实例。SqlSession 可以执行单一的一条 SQL 语句,也可以执行多条,SqlSession 类中定义了一系列执行 SQL 语句的方法,开发人员可以直接操作 SqlSession 实例对象执行映射文件中的 SQL 语句。DbSessionFactory 类中持有 SqlSessionFactory 实例对象,这样 DbSessionFactory 就可以通过 SqlSessionFactory 实例对象获取 SqlSession 实例对象,然后对数据库中的表进行操作。

15.5.1 Session 架构

DbSessionFactory 类中定义了 openSession 方法,该方法的返回值为 Session 类型,下面重点分析 Session,首先从全局的角度讲解 DbSessionFactory、SqlSessionFactory、Session、SqlSession 四者之间的关系,如图 15-5 所示。

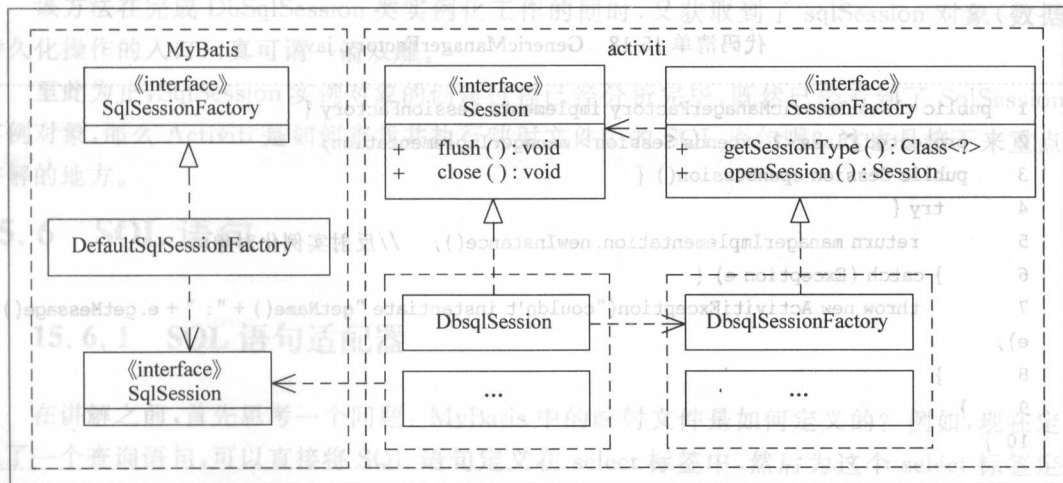


图 15-5 Session 与 SqlSession 功能边界图

通过图 15-5 可以很清晰地看到以上四个种类之间的依赖关系。首先分析 SessionFactory 的一系列子类是如何实现该接口中定义的 openSession 方法。

注意

所有的实体管理类均实现 Session 接口。

15.5.2 实例化方式创建 Session 实例

下面分析实例管理类是如何实现 openSession 方法的。这里所述的用实例化方式创建 Session 实例指的是使用 new 方式,本节以 UserEntityManagerFactory 为例进行探讨,该类的定义如代码清单 15-17 所示。

代码清单 15-17 UserEntityManagerFactory.java

```
1 public class UserEntityManagerFactory implements SessionFactory {
2     public Class<?> getSessionType() {
3         return UserIdentityManager.class; //用户实体管理类
4     }
5     public Session openSession() {
6         return new UserEntityManager(); //实例化用户实体管理类
7     }
8 }
```

该类中 openSession 方法的实现非常简单,直接实例化 UserEntityManager 类, UserEntityManager 类为 Session 接口的实现类。

15.5.3 反射方式创建 Session 实例

反射方式创建 Session 实例只针对通用实体管理类 Session 工厂,即 GenericManagerFactory 类,该类的定义如代码清单 15-18 所示。

代码清单 15-18 GenericManagerFactory.java

```
1 public class GenericManagerFactory implements SessionFactory {
2     protected Class<? extends Session> managerImplementation;
3     public Session openSession() {
4         try {
5             return managerImplementation.newInstance(); //反射实例化对象
6         } catch (Exception e) {
7             throw new ActivitiException("couldn't instantiate " + getName() + ": " + e.getMessage(),
8             e);
9         }
10 }
```

在上述代码中,openSession 方法直接调用 newInstance 方法进行对象的实例化操作。

15.5.4 实例化 DbSqlSession

通过上文的讲解可能会发现一个问题,openSession 方法中并没有看到创建 SqlSession 实例对象的踪迹,如果需要操作数据库中的表,则需要实例化 SqlSession 类,带着这个疑问分析 DbSqlSessionFactory 类中的 openSession 方法的实现逻辑,如代码清单 15-19 所示。

代码清单 15-19 DbSqlSessionFactory.java

```
1 public Session openSession() {  
2     return new DbSqlSession(this);  
3 }
```

openSession 方法直接实例化 DbSqlSession 类,继续分析 DbSqlSession 类的构造方法,相关实现如代码清单 15-20 所示。

代码清单 15-20 DbSqlSession.java

```
1 protected SqlSession sqlSession;  
2 protected DbSqlSessionFactory dbSqlSessionFactory;  
3 public DbSqlSession(DbSqlSessionFactory dbSqlSessionFactory) {  
4     this.dbSqlSessionFactory = dbSqlSessionFactory;  
5     this.sqlSession = dbSqlSessionFactory.getSqlSessionFactory().openSession();  
6 }
```

将以上代码的处理逻辑总结如下。

(1) 第 4 行获取 dbSqlSessionFactory 参数值并将其作为当前类 DbSqlSession 的属性值存在。

(2) 第 5 行通过 dbSqlSessionFactory 对象获取 SqlSessionFactory 实例对象,然后通过该实例对象的 openSession 方法创建 SqlSession 实例。

该方法在完成 DbSqlSession 类实例化工作的同时,又获取到了 sqlSession 对象(数据持久化操作的入口),真可谓一箭双雕。

至此为止,SqlSession 实例对象的创建过程已经分析完毕,既然已经看到了 SqlSession 实例对象,那么 Activiti 是如何查找并执行映射文件中的 SQL 语句呢?这也是接下来重点讲解的地方。

15.6 SQL 语句

15.6.1 SQL 语句适配器

在讲解之前,首先思考一个问题:MyBatis 中的映射文件是如何定义的?例如,现在定义了一个查询语句,可以直接将 SQL 语句定义在 select 标签中,然后为这个 select 标签定义全局唯一的 id 值,如果该 SQL 执行过程中需要外部传入参数,则直接使用 parameterType 属性定义即可,查询的结果可以使用 resultType 或者 resultMap 属性进行

定义,这样程序只需查询标签中的 id 值就可以定位并执行 SQL 语句了。

既然 Activiti 使用 MyBatis 作为数据持久层框架,则映射文件的定义步骤必不可少,思考一个问题:Activiti 可以使用 MySQL、Oracle、也可以使用 PostgreSQL 等,这些不同的数据库 SQL 语法不完全相同,Activiti 如何屏蔽 SQL 语法差异?举个例子,MySQL 和 Oracle 数据库中批量插入的 SQL 语句风格完全迥异,既然 SQL 语法不同,那可以直接在映射文件中定义两个 insert 标签,一个专门针对 MySQL 数据库,一个专门针对 Oracle 数据库,这个问题就迎刃而解。Activiti 确实是这样做的,要是这样设计,Activiti 在运行时就必须获取数据库的类型,进而在操作时才能根据数据库的类型选择不同的 SQL 语句,看到这里就会明白为何流程引擎配置类初始化 dataSource 的过程中需要获取到数据库的类型 databaseType 值,原来 databaseType 值是为这里的操作服务的。

接下来具体分析 Activiti 是如何解决该问题的。在第 15.2.2 节中创建 SqlSessionFactory 实例对象的过程中,设置涉及了 DbSqlSessionFactory.databaseSpecificLimitBeforeStatements.get(databaseType)),接下来以该操作为切入口,分析 Activiti 是如何处理的,如代码清单 15-21 所示。

代码清单 15-21 DbSqlSessionFactory.java

```
1 protected static final Map<String, Map<String, String>> databaseSpecificStatements =
new HashMap<String, Map<String, String>>(); ...//省略一系列的集合
2 static {
3     addDatabaseSpecificStatement("oracle", "bulkInsertProcessDefinition",
4         "bulkInsertProcessDefinition_oracle");
5     ...//省略一系列集合的初始化过程
6 }
```

databaseSpecificStatements 集合为 DbSqlSessionFactory 类中的一个静态变量,因此从该集合中获取数据时,会触发当前类中的第 2~6 行静态代码,第 3~4 行调用 addDatabaseSpecificStatement 方法进行下一步的处理,接下来分析该方法的处理逻辑,如代码清单 15-22 所示。

代码清单 15-22 DbSqlSessionFactory.java

```
1 protected static final Map<String, Map<String, String>> databaseSpecificStatements
2 protected static void addDatabaseSpecificStatement(String databaseType, String activitiStatement,
String.ibatisStatement) {
3     Map<String, String> specificStatements = databaseSpecificStatements.get(databaseType);
4     if (specificStatements == null) {
5         specificStatements = new HashMap<String, String>();
6         databaseSpecificStatements.put(databaseType, specificStatements);
7     }
8     specificStatements.put(activitiStatement,.ibatisStatement);
9 }
```

在上述代码中,第 3 行首先根据数据库类型也就是 databaseType 参数值从 databaseSpecificStatements 集合中进行查询,如果该集合为空,则执行第 4~7 行代码,即初始化集

合并为其添加元素,接着执行第 8 行代码。databaseType 参数的含义表示数据库的类型,activitiStatement 和 ibatisStatement 两个参数值是什么含义呢?在此不妨结合 Activiti 中定义的映射文件进行说明,该文档的相关内容如代码清单 15-23 所示。

代码清单 15-23 org\activiti\db\mapping\entity\ProcessDefinition.xml

```

1 <insert id="bulkInsertProcessDefinition" parameterType="java.util.List">
2     ...//省略 SQL 语句
3 </insert>
4 <insert id="bulkInsertProcessDefinition_oracle" parameterType="java.util.List">
5     ...//省略 SQL 语句
6 </insert>

```

从上面的代码可以看出,第 1~3 行 insert 标签的 id 值为 bulkInsertProcessDefinition,第 4~6 行 insert 标签的 id 值为 bulkInsertProcessDefinition_oracle,以上映射文件中定义的 id 值正是代码清单 15-21 中第 3~4 行操作传递的两个参数值,看到这里的处理恍然大悟,因为这两条 SQL 语句针对不同的数据库,如果现在使用的是 Oracle 数据库则会执行第 4~6 行中的 SQL 语句,使用其他数据库则会执行第 1~3 行中的 SQL 语句。

15.6.2 SQL 执行 id 值生成规则

上文看到了类似<insert id="bulkInsertProcessDefinition">的映射文件标签,为了便于说明,将上述的 id 值统称为 SQL 执行 id 值(在命名空间中值必须是唯一的,可以用来表示定义的 SQL 语句)。众所周知,MyBatis 最终使用 SqlSession 实例对象对数据库中的表进行操作,SqlSession 类中定义了一系列操作数据库的方法,例如 insert、update、delete、select 等,使用这些方法的时候,通常需要传递 SQL 执行 id 值,接下来分析 Activiti 是如何生成 SQL 执行 id 值的,本节以 Group.xml 为例探讨其实现过程,该文件定义了对 ACT_ID_GROUP 表的所有操作,如代码清单 15-24 所示。

代码清单 15-24 org\activiti\db\mapping\entity\Group.xml

```

1 <insert id="insertGroup" parameterType="org.activiti.engine.impl.persistence.entity.
GroupEntity">
2     ...//insert 定义
3 </insert>
4 <insert id="bulkInsertGroup" parameterType="java.util.List">
5     ...//批量插入定义
6 </insert>
7 <update id="updateGroup" parameterType="org.activiti.engine.impl.persistence.entity.
GroupEntity">
8     ...//update 定义
9 </update>
10 <delete id="deleteGroup" parameterType="org.activiti.engine.impl.persistence.entity.
GroupEntity">
11     ...//delete 定义
12 </delete>
13 <select id="selectGroup" parameterType="string" resultMap="groupResultMap">

```



```

14 ...//select 定义
15 </select>

```

接下来分析 Activiti 生成 SQL 执行 id 值的规则,所有的 SQL 执行 id 值操作均定义在 DbSqlSessionFactory 类中,如代码清单 15-25 所示。

代码清单 15-25 DbSqlSessionFactory.java

```

1 public String getInsertStatement(Class<? extends PersistentObject> clazz) {
2     return getStatement(clazz, insertStatements, "insert");    //查询 insert 语句
3 }
4 public String getBulkInsertStatement(Class clazz) {
5     return getStatement(clazz, bulkInsertStatements, "bulkInsert");    //查询 bulkInsert 语句
6 }
7 public String getUpdateStatement(PersistentObject object) {    //查询 update 语句
8     return getStatement(object.getClass(), updateStatements, "update");
9 }
10 public String getDeleteStatement(Class<?> persistentObjectClass) { //查询 delete 语句
11     return getStatement(persistentObjectClass, deleteStatements, "delete");
12 }
13 public String getSelectStatement(Class<?> persistentObjectClass) { //查询 select 语句
14     return getStatement(persistentObjectClass, selectStatements, "select");
15 }

```

上面代码中虽然定义了不同的操作方法,但是最终都是调用 getStatement 方法进行处理。getStatement 方法定义了三个输入参数,第一个参数为 PersistentObject 类运行时的 Class 对象,第二个参数为缓存集合,例如单条数据的插入使用 insertStatements 集合,批量插入使用 bulkInsertStatements 集合等。换言之,也就是每一种不同类型的操作单独使用的集合,第三个参数为 SQL 执行 id 值的前缀。接下来看一下 getStatement 方法的处理逻辑如代码清单 15-26 所示,这里以第 1 行定义的方法为例加以说明。

代码清单 15-26 DbSqlSessionFactory.java

```

1 private String getStatement(Class persistentObjectClass, Map cachedStatements, String prefix) {
2     //首先从缓存中进行获取,如果存在就直接返回
3     String statement = cachedStatements.get(persistentObjectClass);
4     if (statement != null) {
5         return statement;
6     }
7     //prefix 值为 insert, persistentObjectClass.getSimpleName() 值为 GroupEntity
8     statement = prefix + persistentObjectClass.getSimpleName(); //值 insertGroupEntity
9     statement = statement.substring(0, statement.length() - 6); //值 insertGroup
10    cachedStatements.put(persistentObjectClass, statement); //添加到缓存中
11    return statement;
12 }

```

从代码上来看, getStatement 方法的处理经历了如下几个过程。

- (1) 第 3 行尝试从缓存中获取值,如果缓存中存在则第 5 行直接返回。
- (2) 第 8 行生成 statement 值。
- (3) 第 9 行截取 statement 值,截取之后的新值为 insertGroup,这样生成的值就可以和代码清单 15-24 中标签的 id 值对应起来了。
- (4) 将组装之后的结果添加到缓存中。

注意

cachedStatements 为 Map 数据结构,Map 是引用类型,因此存在引用传递的概念。

15.7 数据层和数据的关系

15.7.1 PersistentObject 业务对象

从表 15-1 中了解了映射文件、实体类、数据库表三者之间的关系,那么 Activiti 是如何设计这些实体类的呢?这就涉及了数据是如何从业务层传递到持久层的,用图 15-6 说明更容易理解。



图 15-6 数据传递过程

开发人员只需要组装 PersistentObject 实例对象并将其交给流程引擎,流程引擎根据该实例对象查找其管理类,然后通过管理类执行相应的操作,并最终调用 SqlSession 实例对象完成数据的持久化操作。PersistentObject 类的功能结构图如图 15-7 所示。

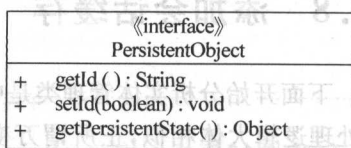


图 15-7 PersistentObject 类的功能结构图

PersistentObject 接口中定义了图 15-7 中的三个方法,表 15-1 中的大部分实体类均实现了该接口。下面讲解这三个方法分别实现的功能,getId、setId 方法比较容易理解,因为几乎所有的数据库表都需要主键值(不包含特殊的表),因此实体类需要定义 id 属性从而与数据库中的主键列进行一一映射,为何需要定义 getPersistentState 方法,getPersistentState 方法完成了什么功能?这就涉及了 Activiti 中的“会话缓存”技术,稍后详细说明,暂且有个印象即可。

15.7.2 实体管理类

上面讲解了几乎所有的实体类均需实现 PersistentObject 接口,PersistentObject 主要用来封装业务数据,那么 Activiti 是如何调度这些实体类进行数据的持久化操作呢?由于 Activiti 中的表非常多,这就导致实体类(也就是映射文件需要操作的类)非常庞大,因此

Activiti 引入了实体管理类,通常情况下实体管理类与实体类是一一对应的,这样既可以分散每个实体管理类的职责,又可以对所有的实体管理类进行一次全局架构,下面分析 Activiti 是如何设计实体管理类的,相关类图如图 15-8 所示。

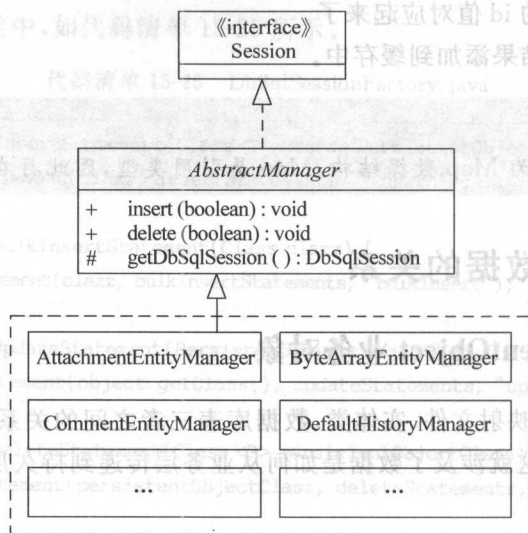


图 15-8 实体管理类

通过图 15-8 可知所有的实体管理类均继承自 `AbstractManager` 类,该类定义了一系列操作 `PersistentObject` 实例对象的方法 `insert`、`delete`、`getDbSqlSession` 等,为何该类中没有定义数据的查询和更新操作呢?当前类中确实没有定义上述两个方法,带着这个疑问分析 `insert` 方法的处理逻辑。

15.8 添加会话缓存

下面开始分析实体管理类是 `insertResource` 方法的处理逻辑。由于实体管理类非常多,而处理逻辑大体相似,正所谓万变不离其宗,这里以资源实体管理类 `ResourceEntityManager` 为例,探讨 `insertResource` 方法的处理流程如代码清单 15-27 所示。

代码清单 15-27 `ResourceEntityManager.java`

```
1 public void insertResource(ResourceEntity resource) {
2     getDbSqlSession().insert(resource);
3 }
```

该方法的处理逻辑非常简单,将其处理过程总结如下。

(1) 第 2 行首先通过 `getDbSqlSession` 方法获取 `DbSqlSession` 实例对象,可是仔细分析发现 `ResourceEntityManager` 类中并没有定义 `getDbSqlSession` 方法,既然该类中没有定义 `getDbSqlSession` 方法,就需要在其父类中进行查找,当前类的父类 `AbstractManager` 如代码清单 15-28 所示。

代码清单 15-28 AbstractManager.java

```

1  protected DbSqlSession getDbSqlSession() {
2      return getSession(DbSqlSession.class);
3  }
4  protected <T> T getSession(Class<T> sessionClass) {
5      return Context.getCommandContext().getSession(sessionClass);
6  }

```

第 2 行直接调用第 4 行定义的方法获取实例对象, 仔细分析第 4 行定义方法的处理逻辑, 该方法直接通过 Context 类获取 CommandContext 实例对象, 然后委托该实例对象中的 getSession 方法进行工作, CommandContext 实例对象既然可以在此处获取, 说明此处是以命令的方式执行该方法的, 因为只有命令才会被命令拦截器链所拦截, 而命令拦截器链正好负责初始化 CommandContext 实例, 这也从侧面验证了 Activiti 确实将客户端一系列的请求组装成命令执行。了解了以上内容之后, 接着分析 CommandContext 实例中的 getSession 方法, 如代码清单 15-29 所示。

代码清单 15-29 CommandContext.java

```

1  public <T> T getSession(Class<T> sessionClass) {
2      Session session = sessions.get(sessionClass);
3      if (session == null) {
4          SessionFactory sessionFactory = sessionFactories.get(sessionClass);
5          if (sessionFactory == null) {
6              throw new ActivitiException("no session factory for " + sessionClass.getName());
7          }
8          session = sessionFactory.openSession();
9          sessions.put(sessionClass, session);
10     }
11     return (T) session;
12 }

```

在调用 getSession 方法时传递的参数值为 DbSqlSession.class, 这一点要时刻牢记, 将该方法的执行逻辑进行如下总结。

- 第 2 行根据 sessionClass 参数值从 sessions 集合中获取 Session 实例对象。
- 如果没有获取到, 则第 4 行从 sessionFactories 集合中进行查找, 如果该操作没有查询到则程序直接报错。
- 如果从 sessionFactories 集合中获取到值, 则第 8~9 行需要重新打开 session 并将其添加到 sessions 集合中。
- 第 11 行返回 session 对象。

(2) DbSqlSession 实例对象获取完毕, 就需要调用该实例对象的 insert 方法进行下一步处理, 如代码清单 15-30 所示。

代码清单 15-30 DbSqlSession.java

```

1  protected Map<Class<? extends PersistentObject>, List<PersistentObject>> insertedObjects

```



```

2 public void insert(PersistentObject persistentObject) {
3     if (persistentObject.getId() == null) {
4         String id = dbSqlSessionFactory.getIdGenerator().getNextId();
5         persistentObject.setId(id);
6     }
7     Class<? extends PersistentObject> clazz = persistentObject.getClass();
8     if (!insertedObjects.containsKey(clazz)) {
9         insertedObjects.put(clazz, new ArrayList<PersistentObject>());
10    }
11    insertedObjects.get(clazz).add(persistentObject);
12    cachePut(persistentObject, false);
13 }

```

该方法中并没有发现任何与数据库交互的操作,暂且将其处理流程总结如下。

(1) 第3行获取 persistentObject 对象的 id 值,如果 id 值为空,则第4~5行需要调用 id 生成器生成新的 id 值并将其填充到该对象中,为何要进行该操作呢?很容易理解,数据库表中的主键值不能为空。

(2) 第7行获取 PersistentObject 类的 Class 实例对象。

(3) 第8行如果 insertedObjects 集合中不存在 clazz 对象,则需要执行第9行代码,该操作是为了确保后续向 insertedObjects 集合添加元素时该集合不为空。

(4) 第11行将 persistentObject 对象添加到 insertedObjects 集合中。

(5) 第12行调用 cachePut 方法缓存 persistentObject 对象,cachePut 方法的相关实现如代码清单 15-31 所示。

代码清单 15-31 DbSqlSession.java

```

1 protected Map<Class<?>, Map<String, CachedObject>> cachedObjects
2 protected CachedObject cachePut(PersistentObject persistentObject, boolean storeState) {
3     Map<String, CachedObject> classCache = cachedObjects.get(persistentObject.getClass());
4     if (classCache == null) {
5         classCache = new HashMap<String, CachedObject>();
6         cachedObjects.put(persistentObject.getClass(), classCache);
7     }
8     CachedObject cachedObject = new CachedObject(persistentObject, storeState);
9     classCache.put(persistentObject.getId(), cachedObject);
10    return cachedObject;
11 }

```

将上面代码的处理过程进行如下总结。

(1) 第3行从 cachedObjects 集合中获取数据,如果没有获取到数据,则第4~7行需要初始化 classCache 集合并将其添加到 cachedObjects 集合中。

(2) 第8行实例化 CachedObject 类,该类的定义如代码清单 15-32 所示。

代码清单 15-32 DbSqlSession.java

```

1 public static class CachedObject {
2     protected PersistentObject persistentObject;

```

```

3    protected Object persistentObjectState;
4    public CachedObject(PersistentObject persistentObject, boolean storeState) {
5        this.persistentObject = persistentObject;
6        if (storeState) {
7            this.persistentObjectState = persistentObject.getPersistentState();
8        }
9    }
10 }

```

CachedObject 类为 DbSqlSession 类中的静态内部类, CachedObject 类的构造方法的执行逻辑非常简单, 首先第 5 行将 persistentObject 参数值设置到 CachedObject 实例对象中, 如果 storeState 参数值为 true, 则执行第 7 行代码, 虽然暂时还不能完全理解这样设计的意图, 但是其执行逻辑可以很轻松的看明白, 对于该类暂且有个印象即可, 稍后加以说明。

(3) 第 9 行将 cachedObject 对象添加到 classCache 集合中。

对于插入数据的 insert 方法, 分析到这里已经看到 Activiti 只需要把期望插入数据库的实体对象放到 DbSqlSession 类中的 insertedObjects 和 cachedObjects 集合中, 好像只是做了缓存, 并没有实质性地调用 SqlSession 实例进行数据的插入操作, 为何这样设计? 实体对象又是在哪里进行入库操作? 稍后统一进行讲解。

15.9 更新操作

Activiti 将数据的更新分为以下两种。

(1) 会话缓存方式更新。

(2) 直接操作 SqlSession 实例更新数据。

15.9.1 会话缓存方式更新

会话缓存的更新方法 update(PersistentObject persistentObject) 的实现逻辑如代码清单 15-33 所示。

代码清单 15-33 DbSqlSession.java

```

1    public void update(PersistentObject persistentObject) {
2        cachePut(persistentObject, false);
3    }

```

该方法仅仅是将 persistentObject 对象添加到缓存中, 并没有真正地进行更新数据库的操作。

15.9.2 SqlSession 方式更新

直接通过操作 SqlSession 实例对象更新数据的方法 update(String statement, Object parameters) 的实现逻辑如代码清单 15-34 所示。

代码清单 15-34 DbSqlSession.java

```

1 public int update(String statement, Object parameters) {
2     String updateStatement = dbSqlSessionFactory.mapStatement(statement);
3     return getSqlSession().update(updateStatement, parameters);
4 }

```

该方法定义了两个参数,其中 statement 参数值对应 MyBatis 映射文件中 update 标签中的 id 值,parameters 参数值为 MyBatis 映射文件中 update 标签执行时需要输入的参数,可以通过该标签中的 parameterType 属性进行定义。将该方法的执行逻辑总结如下。

(1) 第 2 行完成 statement 参数值的校验,statement 参数值就是一个普通的字符串,有什么可校验的价值呢?上文讲解了 DbSqlSessionFactory 类中的 databaseSpecificStatements 集合存储了差异化的 SQL 执行 id 值,该操作就是判断 statement 参数值是否为差异化的 SQL 执行 id 值,如果是就需要获取真实的 SQL 执行 id 值,mapStatement 方法的相关实现如代码清单 15-35 所示。

代码清单 15-35 DbSqlSessionFactory.java

```

1 public String mapStatement(String statement) {
2     if (statementMappings == null) {
3         return statement;
4     }
5     String mappedStatement = statementMappings.get(statement);
6     return (mappedStatement != null ? mappedStatement : statement);
7 }

```

在上述代码中,首先第 2 行判断 statementMappings 集合是否为空,如果该集合为空,则表示不存在差异化的 SQL 执行 id 值,因此不需要处理直接返回 statement 参数值。

statementMappings 集合在 DbSqlSessionFactory 类的 setDatabaseType 方法中进行初始化,如果 statementMappings 集合不为空,则第 5 行从 statementMappings 集合中获取数据,如果没有获取到则直接返回 statement 参数值,否则直接返回获取到的 mappedStatement 值。

(2) 第 3 行调用 SqlSession 实例对象中的 update 方法进行数据的更新操作,该操作执行完毕且事务提交之后没有出现异常,则数据库中的数据更新成功。

15.10 删除操作

对于数据的删除操作来说,Activiti 分为如下两种删除方式。

- (1) 会话缓存方式删除。
- (2) 根据 SQL 执行 id 值和参数直接调用 SqlSession 实例进行删除操作。

15.10.1 DeleteOperation 接口

在学习以上两种实现方式之前首先讲解关于删除操作 Activiti 是如何设计的,Activiti 将删除操作抽象为 DeleteOperation 接口,并提供了不同的实现类,如图 15-9 所示。

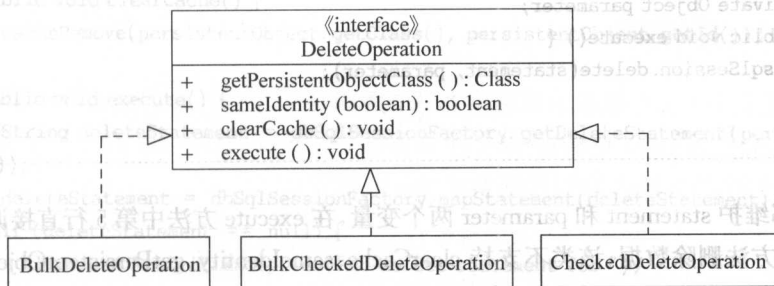


图 15-9 DeleteOperation 接口类图

通过上面的类图可以很清晰地看到 `DeleteOperation` 类的脉络,接下来详细分析类图中类的职责。

(1) `DeleteOperation`: 该接口作为所有删除操作类的父类存在以方便对子类进行统一调度,该接口定义了 `execute` 方法用于执行具体的删除操作逻辑, `clearCache` 方法用于清除缓存中的数据, `sameIdentity` 方法用来比较两个对象是否相等, `getPersistentObjectClass` 方法用来返回 `PersistentObject` 类运行时的 `Class` 对象。

(2) `BulkDeleteOperation`: 该类可以根据 SQL 执行 id 值和 SQL 语句需要的参数值进行删除操作。

(3) `CheckedDeleteOperation`: 该类内部维护 `PersistentObject` 实例对象,并对父类中的方法提供了实现。

(4) `BulkCheckedDeleteOperation`: 该类内部维护 `persistentObjects` 集合(该集合为 `PersistentObject` 类型),并对父类中的方法提供了实现。该类与 `CheckedDeleteOperation` 类的唯一区别在于该类支持批量删除操作。

15.10.2 BulkDeleteOperation 删除数据

对上面每个类的含义有了大致了解之后,接下来分析如何使用 `BulkDeleteOperation` 类进行数据的删除操作,具体实现如代码清单 15-36 所示。

代码清单 15-36 DbSqlSession.java

```

1 protected List<DeleteOperation> deleteOperations = new ArrayList<DeleteOperation>();
2 public void delete(String statement, Object parameter) {
3     deleteOperations.add(new BulkDeleteOperation(statement, parameter));
4 }
  
```

第 3 行根据 `delete` 方法的两个输入参数值实例化 `BulkDeleteOperation` 类,并将该实例对象添加到 `deleteOperations` 集合中, `BulkDeleteOperation` 类的定义如代码清单 15-37 所示。

代码清单 15-37 DbSqlSession.java

```

1 public class BulkDeleteOperation implements DeleteOperation {
2     private String statement;
  
```



```

3     private Object parameter;
4     public void execute() {
5         sqlSession.delete(statement, parameter);
6     }
7 }

```

该类内部维护 statement 和 parameter 两个变量,在 execute 方法中第 5 行直接调用 sqlSession 对象的 delete 方法删除数据,该类不支持 clearCache、sameIdentity、getPersistentObjectClass 操作,因为使用该方式删除数据,直接操作 sqlSession 对象,因此没必要将删除的数据添加到缓存中。

15.10.3 CheckedDeleteOperation 删除数据

DbSqlSession 类中定义的删除方法如代码清单 15-38 所示。

代码清单 15-38 DbSqlSession.java

```

1 public void delete(PersistentObject persistentObject) {
2     for (DeleteOperation deleteOperation: deleteOperations) {
3         if (deleteOperation.sameIdentity(persistentObject)) {
4             return;
5         }
6     }
7     deleteOperations.add(new CheckedDeleteOperation(persistentObject));
8 }

```

该方法第 2~6 行首先循环遍历 deleteOperations 集合,并调用 deleteOperation 对象中的 sameIdentity 方法验证需要删除的数据是否已经存在 deleteOperations 集合中,如果存在则不会执行添加操作,否则第 7 行将 persistentObject 对象添加到 deleteOperations 集合中。需要知道一点,BulkDeleteOperation 类中的 sameIdentity 方法永远返回 false。

CheckedDeleteOperation 类与 BulkCheckedDeleteOperation 类的处理过程类似,两者唯一的区别是 CheckedDeleteOperation 类只支持删除单条数据,而 BulkCheckedDeleteOperation 类支持批量删除,这里以 CheckedDeleteOperation 类的具体实现为例进行详细讲解(CheckedDeleteOperation 类为 DbSqlSession 类中的静态内部类),CheckedDeleteOperation 类的定义如代码清单 15-39 所示。

代码清单 15-39 CheckedDeleteOperation.java

```

1 public class CheckedDeleteOperation implements DeleteOperation {
2     protected final PersistentObject persistentObject;
3     public Class<? extends PersistentObject> getPersistentObjectClass() {
4         return persistentObject.getClass();
5     }
6     public boolean sameIdentity(PersistentObject other) {
7         return persistentObject.getClass().equals(other.getClass())
8             && persistentObject.getId().equals(other.getId());
9     }

```

```
10     public void clearCache() {
11         cacheRemove(persistentObject.getClass(), persistentObject.getId());
12     }
13     public void execute() {
14         String deleteStatement = dbSqlSessionFactory.getDeleteStatement(persistentObject.
15             getClass());
16         deleteStatement = dbSqlSessionFactory.mapStatement(deleteStatement);
17         if (deleteStatement == null) {
18             throw new ActivitiException("no delete statement for ");
19         }
20         if (persistentObject instanceof HasRevision) {
21             int nrOfRowsDeleted = sqlSession.delete(deleteStatement, persistentObject);
22             if (nrOfRowsDeleted == 0) {
23                 throw new ActivitiOptimisticLockingException(persistentObject);
24             }
25         } else {
26             sqlSession.delete(deleteStatement, persistentObject);
27         }
28     }
```

CheckedDeleteOperation 类内部维护 persistentObject 对象,该对象在当前类的构造方法中已经被初始化,对该类中的方法进行如下描述。

- (1) getPersistentObjectClass 方法: 返回 PersistentObject 类运行时的 Class 对象。
- (2) sameIdentity 方法: 负责判断 other 对象与 persistentObject 对象是否相等。
- (3) clearCache 方法: 清除 cachedObjects 集合中的缓存。
- (4) execute 方法: 该方法的执行逻辑比较复杂,下面详细分析。

- 第 14 行获取删除操作的 SQL 执行 id 值。
- 第 15 行获取最终的 SQL 执行 id 值。
- 第 16 行如果 deleteStatement 值为空,则程序直接报错。
- 第 19 行判断 persistentObject 对象是否是 HasRevision 类的实例,如果是则执行第 20 行代码时需要记录删除操作的返回值,如果返回值为 0,表示第 20 行操作并没有删除数据,因此程序直接报错。为何需要判断 persistentObject 对象是不是 HasRevision 类的实例呢? 这就涉及了数据库中为何需要使用乐观锁,乐观锁通常情况下认为数据不会发生冲突,所以在数据更新或者提交时,才会正式检测数据是否冲突,如果发生了冲突,则将错误信息反馈给客户端,让客户端决定下一步该如何处理。下面详细分析 Activiti 是如何使用乐观锁,Activiti 为部分表增加了一个版本标识列即 REV 列,读取数据时将版本列的值一并查询出来,并将其作为数据的一部分,这样当客户端操作数据时,需要判断数据库中的版本值与当前数据的版本值是否一致,如果两者一致,则给予更新操作(更新一次,版本值加 1),否则将当前的数据作为过期数据处理。
- 如果 persistentObject 对象不是 HasRevision 实例对象,既然没有使用到乐观锁,就直接执行第 25 行调用 sqlSession 实例删除数据。

15.10.4 乐观锁

上文讲解了 Activiti 使用乐观锁对数据的版本进行控制,接下来查看映射文件中 delete 语句的定义,由于 Activiti 中的映射文件非常多,本节以 ACT_ID_GROUP 表为例加以说明,该表对应的映射文件内容如代码清单 15-40 所示。

代码清单 15-40 Group.xml

```
1 <delete id="deleteGroup"
2 parameterType="org.activiti.engine.impl.persistence.entity.GroupEntity">
3   delete from ${prefix}ACT_ID_GROUP where ID_=#{id} and REV_=#{revision}
4 </delete>
```

可以看到上面 SQL 语句中添加了 REV_列的判断操作。

15.11 刷新会话缓存入口

接下来详细分析 Activiti 存储会话缓存数据之后,是如何将缓存中的数据刷新到数据库的。在开始学习之前首先回顾一下命令拦截器 CommandContextInterceptor,该类中 execute 方法中 finally 代码块的执行逻辑如代码清单 15-41 所示。

代码清单 15-41 CommandContextInterceptor.java

```
1 CommandContext context = Context.getCommandContext();
2 if (!contextReused) {
3   context.close();
4 }
```

上面代码仅仅是调用 context 对象中的 close 方法,该方法的具体实现如代码清单 15-42 所示。

代码清单 15-42 CommandContext.java

```
1 public void close() {
2   try {
3     try {
4       try {
5         if (exception == null && closeListeners != null) {
6           try {
7             for (CommandContextCloseListener listener : closeListeners) {
8               listener.closing(this);
9             }
10          } catch (Throwable exception) {
11             exception(exception);
12          }
13        }
14        if (exception == null) {
```

```

15         flushSessions();
16     }
17     } catch (Throwable exception) {
18         exception(exception);
19     } finally {
20         try {
21             if (exception == null) {
22                 transactionContext.commit();
23             }
24         } catch (Throwable exception) {
25             exception(exception);
26         }
27         if (exception == null && closeListeners != null) {
28             try {
29                 for (CommandContextCloseListener listener : closeListeners) {
30                     listener.closed(this);
31                 }
32             } catch (Throwable exception) {
33                 exception(exception);
34             }
35         }
36         if (exception != null) {
37             transactionContext.rollback();
38         }
39     } catch (Throwable exception) {
40         exception(exception);
41     } finally {
42         closeSessions();
43     }
44 } catch (Throwable exception) {
45     exception(exception);
46 } finally {
47     }
48 }

```

上面的代码量虽然很多,但是其处理逻辑比较容易理解,该过程涉及了事务的提交(第22行)和回滚(第37行)操作,下面将 close 方法的执行逻辑进行总结。

(1) 第5行如果程序没有出现任何异常情况并且 closeListeners 集合不为空,则循环遍历该集合并在第8行触发 listener 对象的 closing 方法。

(2) 如果程序没有出现异常则第15行执行会话缓存的刷新操作,这个地方就涉及了引擎与数据库的交互,包括数据的入库、删除、更新操作,该步骤非常重要。

(3) 第22行开始提交事务,如果事务提交过程中没有出现异常信息,并且 closeListeners 集合不为空,则循环遍历该集合并在第30行触发 listener 对象的 closed 方法,如果程序出现异常,则第37行进行事务的回滚操作。

(4) 上述一系列操作执行后,第43行调用 closeSessions 方法关闭 SqlSession 实例对象,以防止内存泄露。

15.12 会话缓存数据持久化

介绍了 CommandContext 类中 close 方法的执行逻辑之后,接下来分析会话缓存中的数据是如何被持久化的,flushSessions 方法的定义如代码清单 15-43 所示。

代码清单 15-43 CommandContext.java

```
1 protected void flushSessions() {
2     for (Session session : sessions.values()) {
3         session.flush();
4     }
5 }
```

该方法首先会循环遍历 sessions.values() 值,然后依次调用 session 对象中的 flush 方法进行下一步操作,接下来将重心放到 flush 方法的处理逻辑中如代码清单 15-44 所示。

代码清单 15-44 DbSqlSession.java

```
1 public void flush() {
2     //移除不必要的操作
3     List<DeleteOperation> removedOperations = removeUnnecessaryOperations();
4     flushDeserializedObjects();           //刷新序列化对象
5     List<PersistentObject> updatedObjects = getUpdatedObjects(); //获取需要更新的数据
6     flushInserts();                       //插入
7     flushUpdates(updatedObjects);         //更新
8     flushDeletes(removedOperations);      //删除
9 }
```

为了方便查看,此代码清单中省略了 flush 方法中日志输出的相关代码,下面首先分析 removeUnnecessaryOperations 方法。

15.12.1 移除不必要的数据库

removeUnnecessaryOperations 方法所做的工作就是移除会话缓存中不必要的数据库,举一个简单的例子,上文讲解了 flush 方法对数据的操作顺序依次为插入、更新、删除,这时可能会有一个问题如图 15-10 所示。

例如,Object1 对象既在 cachedObjects 集合(缓存的数据)中,又在 insertedObjects 集合(准备插入数据库的数据)中,这时该如何处理呢?最简单的办法就是首先将 Object1 对象持久化到数据库,然后再将该对象更新到 cachedObjects 集合中,但是这样会引发另一个问题,如果对象持久化到数据库时程序出现了异常,该怎么办呢?更重要的一点是事务操作不是在当前类中进行的,这时该对象怎么可能被成功更新到 cachedObjects 集合呢?这种方案显然不行,Activiti 的做法是,如果对象同时存在于 cachedObjects 和 insertedObjects 两个集合中,就直接将该对象从 cachedObjects 集合中剔除,这样就规避了脏数据的问题,也解决了集合需要更新的问题,可谓两全其美,因为查询数据的时候,首先会从 cachedObjects 集合

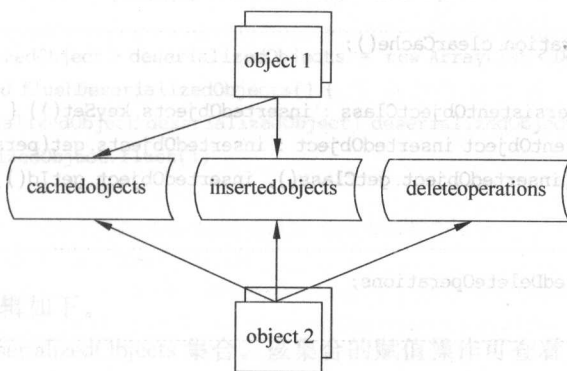


图 15-10 数据的操作

中查询值,如果没有从该集合中查询到,则直接查询数据库,并将查询到的数据添加到 cachedObjects 集合中。

例如, Object2 对象同时存在于 cachedObjects(缓存的数据)、insertedObjects(准备插入数据库的数据)、deleteOperations(准备删除的数据)三个集合,这个时候又该如何处理呢? 既然 Object2 对象在三个集合中都存在,那干脆将该对象从 cachedObjects 和 insertedObjects 集合中进行剔除,这种解决方案既简单又精巧,为什么要这么做呢? 此时此刻,应该牢记 flush 方法对数据的操作顺序依次为插入、更新、删除,试想一下,对于同一个对象首先插入,然后将其更新、最后再将其删除,真的有这个必要吗? 何不将其直接删除。

理解了 Activiti 对于同一个对象存在于多个集合的处理方法,接下来问题就好办多了, removeUnnecessaryOperations 方法的定义如代码清单 15-45 所示。

代码清单 15-45 DbSqlSession.java

```

1  protected List<DeleteOperation> removeUnnecessaryOperations() {
2      List<DeleteOperation> removedDeleteOperations = new ArrayList<DeleteOperation>();
3      for ( Iterator<DeleteOperation> deleteIterator = deleteOperations.iterator();
4            deleteIterator.hasNext(); ) {
5          DeleteOperation deleteOperation = deleteIterator.next();
6          Class deletedPersistentObjectClass = deleteOperation.getPersistentObjectClass();
7          List<PersistentObject> insertedObjectsOfSameClass = insertedObjects.get(deletedPersistentObjectClass);
8          if (insertedObjectsOfSameClass != null && insertedObjectsOfSameClass.size() > 0) {
9              for (Iterator<PersistentObject> insertIterator = insertedObjectsOfSameClass.iterator();
10                    insertIterator.hasNext(); ) {
11                  PersistentObject insertedObject = insertIterator.next();
12                  if (deleteOperation.sameIdentity(insertedObject)) {
13                      insertIterator.remove();
14                      deleteIterator.remove();
15                      removedDeleteOperations.add(deleteOperation);
16                  }
17              }
18          }
19          if (insertedObjects.get(deletedPersistentObjectClass).size() == 0) {
20              insertedObjects.remove(deletedPersistentObjectClass);
21          }
22      }
23  }

```

```

19     }
20     deleteOperation.clearCache();
21 }
22 for (Class persistentObjectClass : insertedObjects.keySet()) {
23     for (PersistentObject insertedObject : insertedObjects.get(persistentObjectClass)) {
24         cacheRemove(insertedObject.getClass(), insertedObject.getId());
25     }
26 }
27 return removedDeleteOperations;
28 }

```

上面代码的执行逻辑包含了上述所说的两种情况,下面对该方法的执行逻辑进行如下总结。

(1) 第 2 行初始化 removedDeleteOperations 集合并作为该方法的返回值,该集合用于存储需要删除的对象。

(2) 第 3~21 行循环遍历 deleteOperations(需要删除的对象)集合。

- 第 5 行通过 deleteOperation 对象中的 getPersistentObjectClass 方法获取 deleted-PersistentObjectClass 对象。
- 第 6 行根据 deletedPersistentObjectClass 对象从 insertedObjects 集合中获取数据,该操作主要是为了验证对象是否既存在于 deleteOperations 集合又存在于 insertedObjects 集合,如果是则继续进行如下处理。
- 循环遍历 insertedObjectsOfSameClass 集合,第 10 行如果 insertedObject 对象与 deleteOperation 对象匹配,那就需要从 insertedObjects 和 deleteOperations 两个集合中分别剔除该对象,并在第 13 行将 deleteOperation 对象添加到 removedDeleteOperations 集合中。
- 第 16~18 行再次确保 deletedPersistentObjectClass 对象不会存在于 insertedObjects 集合中。
- 第 20 行执行 deleteOperation.clearCache() 方法将需要删除的对象从 cachedObjects 集合中剔除。

(3) 第 22~26 行循环遍历 insertedObjects 集合。

该操作就是将既存在于 insertedObjects 集合又存在于 cachedObjects 集合中的对象,从 cachedObjects 集合中剔除。

15.12.2 刷新序列化变量

流程实例运转过程中经常需要使用大量的变量,这些变量的类型可以是 String、Boolean、UUID、Serializable 等,flushDeserializedObjects 方法的职责就是将实现 Serializable 接口的变量(通常情况下以类的方式存在)刷新到数据库,例如定义了一个类 ShareniuSerializable,该类实现了 Serializable 接口,变量的设置形如 map.put("shareniuSerializable", new ShareniuSerializable())。理解了设计意图之后,接下来分析 flushDeserializedObjects 方法的定义如代码清单 15-46 所示。

代码清单 15-46 DbSqlSession.java

```

1 List<DeserializedObject> deserializedObjects = new ArrayList<DeserializedObject>();
2 protected void flushDeserializedObjects() {
3     for (DeserializedObject deserializedObject: deserializedObjects) {
4         deserializedObject.flush();
5     }
6 }

```

该方法的处理逻辑如下。

- (1) 循环遍历 deserializedObjects 集合。该集合的赋值操作可查看 SerializableType 类。
- (2) 第 4 行调用 deserializedObject 对象中的 flush 方法进行数据的刷新操作。

deserializedObject.flush() 方法的相关实现如代码清单 15-47 所示。

代码清单 15-47 DeserializedObject.java

```

1 public void flush() {
2     if (deserializedObject == variableInstanceEntity.getCachedValue() && !variableInstanceEntity.isDeleted()) {
3         byte[] bytes = type.serialize(deserializedObject, variableInstanceEntity);
4         if (!Arrays.equals(originalBytes, bytes)) {
5             Object originalObject = type.deserialize(originalBytes, variableInstanceEntity);
6             byte[] refreshedOriginalBytes = type.serialize(originalObject, variableInstanceEntity);
7             if (!Arrays.equals(refreshedOriginalBytes, bytes)) {
8                 variableInstanceEntity.setBytes(bytes);
9             }
10        }
11    }
12 }

```

第 2 行判断 deserializedObject 对象与 variableInstanceEntity.getCachedValue() 方法的返回值是否相等且 variableInstanceEntity.isDeleted() 方法的返回值是否为 false, 该判断操作主要是为了确保变量值是否被修改或者被删除, 如果变量值被修改了但还没有被删除则开始执行如下的逻辑。

- (1) 第 3 行将 deserializedObject 对象转化为 byte 数组。
- (2) 第 4 行比对 originalBytes 与 bytes 是否相等。
- (3) 第 5 行将 originalBytes 转化为 originalObject 对象, 第 6 行将 originalObject 对象转化为 byte 数组。
- (4) 第 7 行如果 refreshedOriginalBytes 与 bytes 两者不相等, 则说明变量已经被修改了, 那么开始执行第 8 行代码将变量值刷新到会话缓存中。

注意

序列化类型的流程变量最终会存储到 ACT_GE_BYTEARRAY 以及 ACT_RU_VARIABLE 表中。

15.12.3 获取更新对象

getUpdatedObjects 方法负责获取需要更新的对象,在 DbSqlSession 类中并没有单独提供存储更新对象的集合,Activiti 为什么这样设计呢?下面分析 getUpdatedObjects 方法,该方法的具体实现如代码清单 15-48 所示。

代码清单 15-48 DbSqlSession.java

```

1 public List<PersistentObject> getUpdatedObjects() {
2     List<PersistentObject> updatedObjects = new ArrayList<PersistentObject>();
3     for (Class<?> clazz: cachedObjects.keySet()) {
4         Map<String, CachedObject> classCache = cachedObjects.get(clazz);
5         for (CachedObject cachedObject: classCache.values()) {
6             PersistentObject persistentObject = cachedObject.getPersistentObject();
7             if (!isPersistentObjectDeleted(persistentObject)) {
8                 Object originalState = cachedObject.getPersistentObjectState();
9                 if (persistentObject.getPersistentState() != null &&
10                     !persistentObject.getPersistentState().equals(originalState)) {
11                     updatedObjects.add(persistentObject);
12                 }
13             }
14         }
15     }
16     return updatedObjects;
17 }

```

该方法的处理逻辑如下。

(1) 第 2 行初始化 updatedObjects 集合(存储需要更新的对象)并作为当前方法的返回值。

(2) 遍历 cachedObjects 集合。

第 4 行根据 clazz 对象从 cachedObjects 集合中获取值,如果没有获取到则表明没有需要更新的对象,如果获取到则进行下一步的处理。

(3) 确定对象是否有必要更新。首先第 6 行通过 cachedObject 对象获取持久化对象 persistentObject,第 7 行调用的 isPersistentObjectDeleted 方法判断 persistentObject 对象是否存在于 deleteOperations 集合中,如果存在该集合中,就不会进行下一步的处理,因为对象马上就要删除了,更新已经没有意义了,否则继续进行下一步处理,下一步的处理逻辑比较难理解但是非常关键,下面详细分析。

- 第 8 行根据 cachedObject 对象获取 originalState 对象,上文讲解了实例化 CachedObject 类的时候需要传递两个参数,其中第二个参数为 storeState(boolean 类型),只有该参数值为 true 时,CachedObject 实例对象中的 persistentObjectState 属性值才不为空,否则 persistentObjectState 变量值为空。
- 如果 persistentObject.getPersistentState() 方法的返回值不为空且与 originalState 对象不匹配,则将 persistentObject 对象添加到 updatedObjects 集合中,否则不会将该对象添加到 updatedObjects 集合中。

下面详细分析 `getPersistentState()` 方法,因为几乎所有的实体类都实现了 `PersistentObject` 接口,所以这里以 `GroupEntity` 实体类为例,探讨该方法的实现逻辑,如代码清单 15-49 所示。

代码清单 15-49 `GroupEntity.java`

```
1 public Object getPersistentState() {
2     Map<String, Object> persistentState = new HashMap<String, Object>();
3     persistentState.put("name", name);
4     persistentState.put("type", type);
5     return persistentState;
6 }
```

该方法只是在第 2 行初始化了 `persistentState` 集合,然后第 3~4 行为该集合添加当前类中的部分属性值。下面使用图 15-11 对该过程进行通俗易懂的描述。

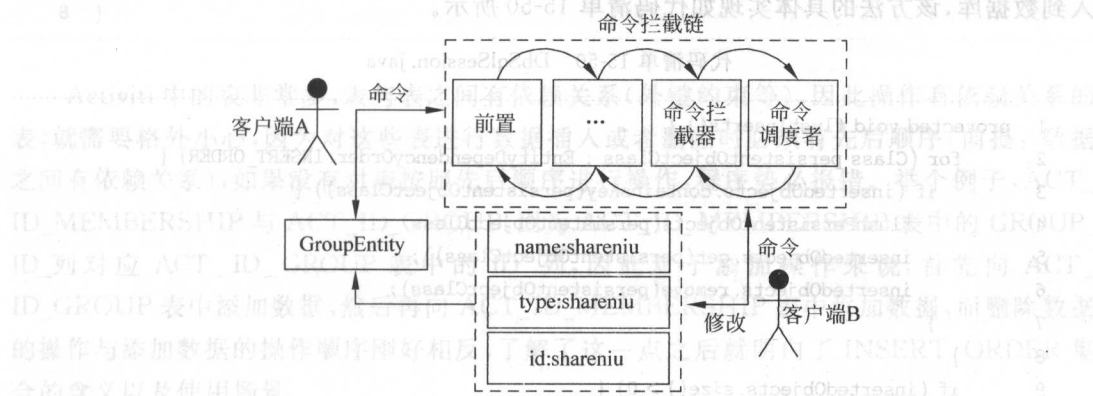
图 15-11 `getPersistentState` 方法返回值的示意图

图 15-11 中假如客户端 A 发送了一个命令,将 `GroupEntity` 实例对象添加到数据库,该命令经过一系列的命令拦截链之后,命令开始原路返回,并且内存中已经存储了 `GroupEntity` 实例对象(将其称之为原值),当执行到命令拦截器之前且程序还没有开始进行 `flushSessions()` 动作时,客户端 A 或者其他的客户端 B 对 `GroupEntity` 实例中的属性值进行了修改,例如修改了 `name` 值为 `shareniu`,那么当客户端 A 执行 `flushSessions` 方法时,因为 `GroupEntity` 实例对象中的属性值与原值不一致,则会触发更新操作。

假如同样地修改操作,客户端 B 修改了该实例对象中的 `id` 值,思考一下会发生更新操作吗?显然不会,为什么呢?因为 `GroupEntity` 类中 `getPersistentState` 方法的返回对象中并没有对 `id` 值进行定义,因此修改 `id` (`getPersistentState` 方法中之前没有定义的属性)值不会触发更新操作。更新操作判断的核心点在于只关心实体类中 `getPersistentState` 方法所定义的属性,其他属性是否变化一概不理睬,Activiti 这样设计存在如下两个缺陷。(2)

(1) 例如,对于 `id` (通常对应数据库中的主键)值来说,如果 Activiti 运行在一个高并发的场景下,那么就有可能出问题,又如客户端 A 生成了一个主键 `id` 值,程序还没有来得及进行 `flushSessions` 操作,这时客户端 B 又需要生成一个主键 `id` 值,这个 `id` 值就有可能被占用了,由于数据库中的主键值不能相同,因此通常情况下,并发量非常大的场景中主键 `id` 建议

使用 UUID 生成策略。

(2) 会话缓存中的数据在没有刷新到数据库之前,如果系统宕机,则会话缓存中的数据丢失,因此使用该功能有风险,因为引擎并没有提供容错机制。

通过上面的分析可知只有 PersistentObject 接口中 getPersistentState 方法定义的属性值发生了变化,Activiti 才会去更新对象,这种方式对于客户端来说,可能有点不灵活,如果开发人员不打算使用该方式更新对象,Activiti 也提供了 update(String statement, Object parameters)方法,可以直接通过 SqlSession 实例对象更新数据。

Activiti 自身的会话缓存机制设计简单,性能稳定,但有局限性,不适用于复杂场景,在实际项目开发中,如果业务场景比较复杂可以结合 MyBatis 中的缓存机制使用。

15.12.4 刷新数据

flushInserts 方法主要用于操作 insertedObjects 集合中的数据,并将该集合中的数据插入到数据库,该方法的具体实现如代码清单 15-50 所示。

代码清单 15-50 DbSqlSession.java

```

1  protected void flushInserts() {
2      for (Class persistentObjectClass : EntityDependencyOrder. INSERT_ORDER) {
3          if (insertedObjects.containsKey(persistentObjectClass)) {
4              flushPersistentObjects(persistentObjectClass,
5                  insertedObjects.get(persistentObjectClass));
6              insertedObjects.remove(persistentObjectClass);
7          }
8      }
9      if (insertedObjects.size() > 0) {
10         for (Class persistentObjectClass : insertedObjects.keySet()) {
11             flushPersistentObjects(persistentObjectClass,
12                 insertedObjects.get(persistentObjectClass));
13         }
14     }
15     insertedObjects.clear();
16 }

```

将 flushInserts 方法的处理逻辑进行如下总结。

(1) 第 2~8 行循环遍历 EntityDependencyOrder. INSERT_ORDER 集合,如果 insertedObjects 集合中包含该集合中的任意一个元素,那么执行第 4~5 行调用 flushPersistentObjects 方法将数据插入到数据库中,该操作执行完后第 6 行从 insertedObjects 集合中剔除该对象。

(2) 上述步骤执行完毕,第 9 行如果 insertedObjects 集合还存在值,则继续调用 flushPersistentObjects 方法进行处理,看到这里可能会有疑问:步骤(1)已经将集合中的数据插入数据库了,怎么这里又进行了一次判断呢?可以推测这个地方的对象可能没有在 EntityDependencyOrder. INSERT_ORDER 集合中进行定义,或者是开发人员自定义的。

(3) 第 15 行清空 insertedObjects 集合中的数据。

15.12.5 解决依赖数据插入先后顺序

根据上面对 flushInserts 方法处理逻辑的分析可知,该方法首先需要遍历 EntityDependencyOrder.INSERT_ORDER 集合,下面详细分析该集合的定义以及初始化过程,如代码清单 15-51 所示。

代码清单 15-51 EntityDependencyOrder.java

```
1 public static List<Class> DELETE_ORDER = new ArrayList<Class<? extends PersistentObject>>();
2 public static List<Class> INSERT_ORDER = new ArrayList<Class<? extends PersistentObject>>();
3 static {
4     DELETE_ORDER.add(PropertyEntity.class);
5     ...//省略一系列的添加过程
6     INSERT_ORDER = new ArrayList<Class<? extends PersistentObject>>(DELETE_ORDER);
7     Collections.reverse(INSERT_ORDER);
8 }
```

Activiti 中的表非常多,表与表之间有依赖关系(外键约束等),因此操作有依赖关系的表,就需要格外小心,因为对这些表进行数据插入或者删除时必须要有先后顺序(前提:数据之间有依赖关系),如果没有对表按照先后顺序进行操作,程序势必报错。举个例子,ACT_ID_MEMBERSHIP 与 ACT_ID_GROUP,因为 ACT_ID_MEMBERSHIP 表中的 GROUP_ID_列对应 ACT_ID_GROUP 表中的 ID_列,因此对于添加操作来说,首先向 ACT_ID_GROUP 表中添加数据,然后再向 ACT_ID_MEMBERSHIP 表中添加数据,而删除数据的操作与添加数据的操作顺序刚好相反,了解了这一点之后就明白了 INSERT_ORDER 集合的含义以及使用场景。

接下来分析 flushPersistentObjects 方法,该方法的具体实现如代码清单 15-52 所示。

代码清单 15-52 DbSqlSession.java

```
1 protected void flushPersistentObjects(Class persistentObjectClass, List<PersistentObject>
persistentObjectsToInsert) {
2     if (persistentObjectsToInsert.size() == 1) {
3         flushRegularInsert(persistentObjectsToInsert.get(0), persistentObjectClass);
4     } else if (Boolean.FALSE.equals(dbSqlSessionFactory.isBulkInsertable(persistentObjectClass))) {
5         for (PersistentObject persistentObject : persistentObjectsToInsert) {
6             flushRegularInsert(persistentObject, persistentObjectClass);
7         }
8     } else {
9         flushBulkInsert(persistentObjectsToInsert.get(persistentObjectClass), persistentObjectClass);
10    }
11 }
```

首先第 2 行判断 persistentObjectsToInsert 集合的大小,如果该集合中只有一个元素或者该集合中存在多个元素但是不支持批量操作,则需要循环遍历该集合并在第 6 行调用 flushRegularInsert 方法进行单条数据的插入,否则第 9 行调用 flushBulkInsert 方法进行批

量插入操作。

扩展

DbSessionFactory 类中的 bulkInsertableMap 集合初始化过程可以参考该类的 initBulkInsertEnabledMap 方法。

flushRegularInsert 方法的实现如代码清单 15-53 所示。

代码清单 15-53 DbSessionFactory.java

```
1 protected void flushRegularInsert(PersistentObject persistentObject, Class <? extends
PersistentObject> clazz) {
2     String insertStatement = dbSessionFactory.getInsertStatement(persistentObject);
3     insertStatement = dbSessionFactory.mapStatement(insertStatement);
4     sqlSession.insert(insertStatement, persistentObject);
5     if (persistentObject instanceof HasRevision) {
6         ((HasRevision) persistentObject).setRevision(((HasRevision)
persistentObject).getRevisionNext());
7     }
8 }
9 }
```

在上述代码中,第 2 行构造了 SQL 语句的执行 id 值,第 3 行取出最终 SQL 语句执行 id 值,第 4 行调用 sqlSession 对象进行数据的入库操作,上述操作执行完毕,如果 persistentObject 是 HasRevision 实例对象,则需要设置该对象的 revision 值(乐观锁),该值默认为 1,可以参考映射文件中 insert 标签定义的 SQL 语句进行查看。

由于本书篇幅有限,flushUpdates、flushBulkInsert 和 flushDeletes 方法的实现可以结合上述的讲解自行学习。

15.12.6 性能优化

对于优化 Activiti 操作数据库的性能来说,可以分为如下几个方面。

(1) 开启批量插入数据的功能,可以设置 isBulkInsertEnabled 开关属性值为 true。

(2) 增加数据批量插入条数的阈值,可以设置 maxNrOfStatementsInBulkInsert 开关属性值为 1000,甚至更多,该值默认为 100。

第 16 章

实 战

前面章节重点讲解了 Activiti 框架的内部实现原理，了解了原理之后，本章从实战的角度出发，第一可以对前面的章节做一次回顾；第二可以完全接管 Activiti，从而使开发人员可以灵活地应对需求以及扩展 Activiti。

16.1 高并发 id 生成器

16.1.1 id 生成器初始化

15.12.3 节中讲解到如果 Activiti 运行在高并发的环境下，极有可能生成相同的 id 值（数据库中的主键 ID_列），本节主要分析 id 生成器的初始化过程，如代码清单 16-1 所示。

代码清单 16-1 ProcessEngineConfigurationImpl.java

```
1 protected IdGenerator idGenerator;
2 protected void initIdGenerator() {
3     if(idGenerator == null) {
4         CommandExecutor idGeneratorCommandExecutor = null;
5         if(idGeneratorDataSource != null) {
6             ProcessEngineConfigurationImpl processEngineConfiguration = new StandaloneProcess-
7             EngineConfiguration();
8             processEngineConfiguration.setDataSource(idGeneratorDataSource);
9             processEngineConfiguration.setDatabaseSchemaUpdate(DB_SCHEMA_UPDATE_FALSE);
10            processEngineConfiguration.init();
11            idGeneratorCommandExecutor = processEngineConfiguration.getCommandExecutor();
12        } else if(idGeneratorDataSourceJndiName != null) {
13            ...//获取 idGeneratorCommandExecutor 对象
14        } else {
15            idGeneratorCommandExecutor = getCommandExecutor();
16        }
17    }
18 }
```

```

16      DbIdGenerator dbIdGenerator = new DbIdGenerator();
17      dbIdGenerator.setIdBlockSize(idBlockSize);
18      dbIdGenerator.setCommandExecutor(idGeneratorCommandExecutor);
19      dbIdGenerator.setCommandConfig(getDefaultCommandConfig().transactionRequiresNew
    ());
20      idGenerator = dbIdGenerator;
21  }
22 }

```

将该方法的处理过程进行如下总结。

(1) 第 3 行判断 idGenerator 开关属性值是否为空,如果不为空则程序不予处理,否则进行如下的处理。

(2) 第 5 行如果 idGeneratorDataSource 开关属性值不为空,则开始执行如下逻辑。

- 第 6 行实例化 ProcessEngineConfigurationImpl 类。
- 第 7~8 行为 processEngineConfiguration 对象填充属性,填充的属性有 dataSource、databaseSchemaUpdate。
- 第 9 行初始化流程引擎配置类。

• 第 10 行通过 processEngineConfiguration 对象获取命令执行器 CommandExecutor 实例对象。

(3) 如果 idGeneratorDataSource 开关属性值为空但 idGeneratorDataSourceJndiName 开关属性值不为空,则执行第 12 行代码,该过程可以参考步骤(2)。

(4) 以上两种情况都不符合,则第 14 行调用 getCommandExecutor() 方法获取命令执行器 CommandExecutor 实例对象。

(5) 第 16 行实例化 DbIdGenerator 类。

(6) 第 17~19 行为 dbIdGenerator 填充,有属性值 idBlockSize、commandExecutor、commandConfig。

对于 idGeneratorDataSource 和 idGeneratorDataSourceJndiName 两个开关属性来说,前者使用 dataSource 方式配置任意数据源,后者使用 JNDI 方式配置数据源。

注意

idBlockSize 开关属性默认值为 2500。

16.1.2 自增 id 生成器

上面讲解了 Activiti 默认使用 DbIdGenerator 类作为 id 生成器,该类的相关定义如代码清单 16-2 所示。

代码清单 16-2 DbIdGenerator.java

```

1 public synchronized String getNextId() {
2     if (lastId < nextId) {
3         getNewBlock();
4     }

```

```

5 long _nextId = nextId++;
6 return Long.toString(_nextId);
7 }
8 protected synchronized void getNewBlock() {
9     IdBlock idBlock = commandExecutor.execute(commandConfig, new GetNextIdBlockCmd(idBlockSize));
10    this.nextId = idBlock.getNextId();
11    this.lastId = idBlock.getLastId();
12 }

```

第2行如果 lastId 值小于 nextId 值,则执行第3行调用 getNewBlock()方法,否则执行第5~6行代码。该操作的目的是为了控制程序访问数据库的频率,以提升性能。getNewBlock()方法的处理逻辑如下。

(1) 第9行通过 commandExecutor 对象执行 GetNextIdBlockCmd 命令(需要传入输入参数 idBlockSize)。

(2) 第10~11行设置 DbIdGenerator 实例对象的 nextId 和 lastId 属性值。

GetNextIdBlockCmd 命令类的相关定义如代码清单 16-3 所示。

代码清单 16-3 GetNextIdBlockCmd.java

```

1 public IdBlock execute(CommandContext commandContext) {
2     PropertyEntity property = (PropertyEntity) commandContext.getPropertyEntityManager().
3     .findPropertyById("next.dbid");
4     long oldValue = Long.parseLong(property.getValue());
5     long newValue = oldValue + idBlockSize;
6     property.setValue(Long.toString(newValue));
7     return new IdBlock(oldValue, newValue - 1);
8 }

```

该方法的处理过程如下。

(1) 第2~3行通过 commandContext 获取 PropertyEntityManager 实例对象(负责管理 ACT_GE_PROPERTY 表),然后调用 PropertyEntityManager 实例对象中的 findPropertyById 方法获取 PropertyEntity 实例对象。其中,"next.dbid"值对应 ACT_GE_PROPERTY 表 NAME_列的值。

(2) 第4~5行计算 oldValue 和 newValue 值。

(3) 第6行将 newValue 值设置到 property 对象中,该操作的目的是将新生成的 newValue 值更新到 ACT_GE_PROPERTY 表。

(4) 第7行实例化 IdBlock 类。

16.1.3 自定义主键生成器

经过上文的详细讲解可知 DbIdGenerator 类生成 id 值存在如下两个缺陷。

- (1) 如果 idBlockSize 开关属性值比较小则会频繁查询数据库,效率低下。
- (2) 高并发的情况下可能会造成主键值冲突。

在实际项目开发中通常使用 UUID 生成器, Activiti 内置了 UUID 生成器, 即 StrongUuidGenerator 类, 开发人员可以通过操作流程引擎配置类实现 UUID 生成器的注入, 整个过程如代码清单 16-4 所示。

代码清单 16-4 activiti.cfg.xml

```

1 <bean id="processEngineConfiguration"
2 class="org.activiti.spring.SpringProcessEngineConfiguration">
3   <property name="idGenerator">
4     <bean class="org.activiti.engine.impl.persistence.StrongUuidGenerator"/>
5   </property>
6 </bean>

```

引擎提供的 StrongUuidGenerator 类依赖 FastXml 框架中的 uuid 模块, 因此使用时, 需要引入相关的 jar 包, 本书中使用的程序包为 Java-uuid-generator-3.1.3.jar。

扩展

如果觉得 StrongUuidGenerator 类处理不够友好, 可以自定义一个 id 生成器并实现 IdGenerator 接口, 然后将其注入流程引擎配置类。

16.2 变量类型

流程实例运转过程中, 变量的使用是必不可少的一个环节, 变量的类型有很多种, 最常用的有 boolean、long、uuid 等, Activiti 是如何处理这些变量? 又是如何获取变量的类型呢? 如果觉得 Activiti 中的变量类型不够丰富, 有没有一种方法实现自定义变量类型呢? 带着上述一系列问题开启新的学习之旅。

16.2.1 初始化变量管理类

首先讲解变量管理类的初始化过程, 如代码清单 16-5 所示。

代码清单 16-5 ProcessEngineConfigurationImpl.java

```

1 protected void initVariableTypes() {
2   if (variableTypes == null) {
3     variableTypes = new DefaultVariableTypes();
4     if (customPreVariableTypes != null) {
5       for (VariableType customVariableType: customPreVariableTypes) {
6         variableTypes.addType(customVariableType);
7       }
8     }
9     variableTypes.addType(new NullType());
10    variableTypes.addType(new StringType(getMaxLengthString()));
11    variableTypes.addType(new LongStringType(getMaxLengthString() + 1));
12    ...//省略其他的变量处理类
13    if (customPostVariableTypes != null) {

```

```

14         for (VariableType customVariableType: customPostVariableTypes) {
15             variableTypes.addType(customVariableType);
16         }
17     }
18 }
19 }

```

将该方法的处理过程总结如下。

(1) 第2行如果 variableTypes 开关属性值不为空,则程序不予处理,否则开始执行如下逻辑。

(2) 第3行实例化 DefaultVariableTypes 类。

(3) 第4行如果 customPreVariableTypes 开关属性值不为空,则开始循环遍历 customPreVariableTypes 集合,并通过 variableTypes 对象中的 addType 方法将该集合中的元素添加进去。

(4) 第9~12行添加一系列的内置变量处理类。例如 NullType、StringType 等。

(5) 如果开关属性 customPostVariableTypes 不为空,则开始循环遍历 customPostVariableTypes 集合,并通过 variableTypes 对象中的 addType 方法将该集合中的元素添加进去。

在上述代码中,执行第10行操作时调用了 getMaxLengthString 方法,该方法的定义如代码清单 16-6 所示。

代码清单 16-6 ProcessEngineConfigurationImpl.java

```

1  protected int getMaxLengthString() {
2      if (maxLengthStringVariableType == -1) {
3          if ("oracle".equalsIgnoreCase(databaseType) == true) {
4              return DEFAULT_ORACLE_MAX_LENGTH_STRING;           //2000
5          } else {
6              return DEFAULT_GENERIC_MAX_LENGTH_STRING;           //4000
7          }
8      } else {
9          return maxLengthStringVariableType;
10     }
11 }

```

在上述代码中,第2行如果 maxLengthStringVariableType 开关属性值为-1,则流程引擎连接的数据库类型为 Oracle 时长度为 2000,其他类型的数据库长度为 4000。

variableTypes.addType 方法的具体实现,如代码清单 16-7 所示。

代码清单 16-7 DefaultVariableTypes.java

```

1  private final List<VariableType> typesList = new ArrayList<VariableType>();
2  private final Map<String, VariableType> typesMap = new HashMap<String, VariableType>();
3  public DefaultVariableTypes addType(VariableType type) {
4      return addType(type, typesList.size());
5  }
6  public DefaultVariableTypes addType(VariableType type, int index) {

```

```

7     typesList.add(index, type);
8     typesMap.put(type.getTypeName(), type);
9     return this;
10 }

```

第3行定义的方法直接委托第6行定义的方法进行工作,第6行定义的方法首先将 type 对象添加到 typesList 集合中,然后再将 type 对象添加到 typesMap 集合中,typesMap 集合为 Map 数据结构,key 为 type 对象的 getTypeName 方法返回值,value 为 type 对象。

通过该方法的处理逻辑可知,通过设置 customPreVariableTypes 开关属性可以覆盖系统内置变量处理类。customPostVariableTypes 则给开发人员最后一次机会修改变量处理类。

16.2.2 变量管理类架构

DefaultVariableTypes 类实现了 VariableTypes 接口,该类为变量管理类,其内部持有所有的变量处理类。该接口的类图如图 16-1 所示。

简单介绍 VariableTypes 接口中定义的方法。

(1) getVariableType: 根据变量类型获取 VariableType 实例对象。

(2) addType(VariableType type)与 addType(VariableType type, int index)方法结合起来使用,主要用于添加变量处理类。

(3) getTypeIndex: 获取变量处理类在 typesList 集合中的位置。

(4) findVariableType: 查找变量处理类。对于 DefaultVariableTypes 类中的实现来说,首先遍历 typesList 集合,如果查找到了变量处理类,该方法直接返回,否则程序报错。通过这里的处理逻辑,可以得出一个结论,对于同种类型的变量来说,该变量处理类在集合中的位置越靠前,则使用优先级越高。

(5) removeType: 移除变量处理类。

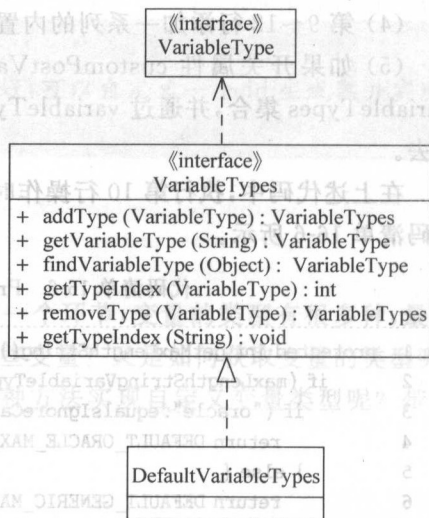


图 16-1 VariableTypes 类图

建议

可以结合 DefaultVariableTypes 类进行学习。

16.2.3 变量处理类

DefaultVariableTypes 类负责管理所有的变量处理类,Activiti 是如何设计这些变量处理类呢?所有的变量处理类均需要实现 VariableType 接口,该接口的定义如代码清单 16-8 所示。

代码清单 16-8 VariableType.java

```
1 public String getTypeName();
2 boolean isCachable();
3 boolean isAbleToStore(Object value);
4 void setValue(Object value, ValueFields valueFields);
5 Object getValue(ValueFields valueFields);
```

(1) `getTypeName`: 获取变量类型的名称, 例如 `BooleanType` 类中该方法的返回值为“boolean”。

(2) `isCachable`: 变量是否支持缓存功能。

(3) `isAbleToStore`: 变量处理类是否有能力处理指定的变量, 例如 `BooleanType` 类只负责处理 `Boolean` 类型的变量。

(4) `setValue`: 为 `valueFields` 对象填充属性值。对应 `ACT_RU_VARIABLE` 表中的 `TYPE_` 列。

(5) `getValue`: 从 `valueFields` 对象中获取值。对应 `ACT_RU_VARIABLE` 表中的 `TYPE_` 列。

注意

`ValueFields` 接口的实现类之一是 `VariableInstanceEntity` 类, 该类为 `ACT_RU_VARIABLE` 表对应的实体类。

16.2.4 自定义变量处理类

默认情况下, Activiti 根据变量的名称查询该变量的类型时, 会通过变量管理类依次查询所有的变量处理类, 直到查找到为止。如果开发人员没有定义 `customPostVariableTypes` 开关属性值, 则 `SerializableType` 类用于处理序列化变量, 这种做法是可取的, 但是有如下几个缺点。

(1) 开发人员必须谨慎使用 `serialVersionUID` 字段对象。

(2) 对于重构不友好, 改变一个类或字段名称就需要重新修改源代码。

(3) 序列化的对象是无法进行数据库查询操作的。

在实际项目开发中, 可以把 `JSON` 字符串存储到 `ACT_RU_VARIABLE` 表中的 `text_` 字段, 并可以通过数据库直接查询, 但这种实现方式受限于字符串的最大长度, 默认情况下 Activiti 定义为 4000 个字符。

首先定义一个变量信息承载类如代码清单 16-9 所示。

代码清单 16-9 Shareniu.java

```
1 public class Shareniu implements Serializable {
2     private String id;
3     private String name;
4     private int age;
5 }
```


下面重点讲解自定义变量处理类的实现过程,如代码清单 16-10 所示。

代码清单 16-10 ShareniuAsJsontype.java

```

1 public class ShareniuAsJsontype implements VariableType {
2     ObjectMapper mapper = new ObjectMapper();
3     public String getTypeName() {
4         return "shareniu"; // 变量类型为 shareniu,自定义的名称可以随意修改
5     }
6     public boolean isCachable() {
7         return true; // 开启缓存,提升效率
8     }
9     // 如果是 value 是 ShareniuAsJsontype 类或者 ShareniuAsJsontype 的子类均可以处理
10    public boolean isAbleToStore(Object value) {
11        if (value == null) {
12            return true;
13        }
14        return Shareniu.class.isAssignableFrom(value.getClass());
15    }
16    public void setValue(Object value, ValueFields valueFields) {
17        if (null == value) {
18            valueFields.setTextValue("");
19        } else {
20            try {
21                valueFields.setTextValue(mapper.writeValueAsString(value));
22            } catch (IOException ioe) {
23            }
24        }
25    }
26    public Object getValue(ValueFields valueFields) {
27        try {
28            return mapper.readValue(valueFields.getTextValue(), Shareniu.class);
29        } catch (IOException ioe) {
30            return null;
31        }
32    }
33 }

```

注意

ShareniuAsJsontype 类中使用了 ObjectMapper 类,该类位于 jackson-databind-2.5.0.jar 中。

上述准备工作完毕需要将自定义变量处理类注入流程引擎配置类,如代码清单 16-11 所示。

代码清单 16-11 activiti.cfg.xml

```

1 <bean id="processEngineConfiguration"
2 class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

```

```
3 <property name="customPreVariableTypes">
4 <list>
5 <bean class="com.shareniu.chapter16.variabletype.ShareniuAsJsontype" />
6 </list>
7 </property>
8 </bean>
```

为了方便测试可以启动任意一个流程实例,然后设置 Shareniu 类型的变量如代码清单 16-12 所示。

代码清单 16-12 VariabletypeTest.java

```
public void startProcessInstanceById(){
    Map<String, Object> variables = new HashMap<String, Object>();
    variables.put("shareniu", new Shareniu("1", "shareniu1",18));
    runtimeService.startProcessInstanceById("myProcess:1:4",variables);
}
```

上述代码执行完毕 ACT_RU_VARIABLE 表中新增的数据如图 16-2 所示。

ID_	REV_	TYPE_	NAME_	TEXT_
27503	1	shareniu	shareniu	{"id":"1","name":"shareniu1","age":18}

图 16-2 ACT_RU_VARIABLE 表中新增的数据

16.3 ServiceLoader 方式注入配置器

在使用配置器时,可以使用 java.util.ServiceLoader 类从配置文件中加载 ProcessEngineConfigurator 接口的实现类,该方式主要根据 META-INF/services 目录下的指定文件加载 ProcessEngineConfigurator 接口的实现类,然后返回该实例对象。首先定义一个配置器如代码清单 16-13 所示。

代码清单 16-13 ShareniuConfigurator.java

```
1 public class ShareniuConfigurator extends AbstractProcessEngineConfigurator {
2     public void beforeInit(ProcessEngineConfigurationImpl processEngineConfiguration) {
3         System.out.println("Shareniu:beforeInit");
4     }
5     public void configure(ProcessEngineConfigurationImpl processEngineConfiguration) {
6         System.out.println("Shareniu:configure");
7     }
8 }
```

配置器定义之后,需要新建 META-INF/services 目录,然后在该目录下新建一个文件,文件的名称为 ProcessEngineConfigurator 类的全名,文件的内容为 ShareniuConfigurator 类的全名,如图 16-3 所示。

可以参考第 2.5 节的讲解进行测试。

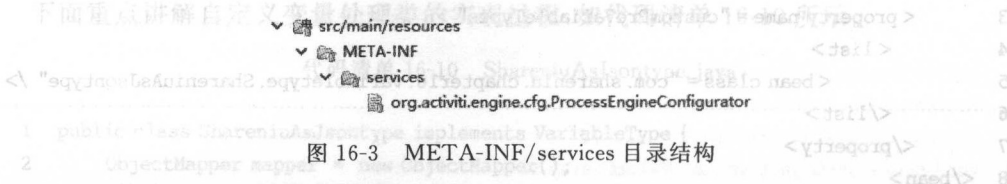


图 16-3 META-INF/services 目录结构

扩展

可以结合上述案例学习 activiti-jmx-5.21.0 模块中的 JMXConfigurator 配置器。

16.4 节点跳转

考虑这样一个实际应用——商品采购系统。供应商上新商品时,需进行商务审核,商务审核通过则提交给运营审核,商务审核失败则退回供应商,运营审核成功后提交合同签订,运营审核失败则退回商务审核或者直接退回供应商,合同签订审核通过则流程实例结束,合同签订审核失败则返回运营审核或者退回商务审核,或者退回供应商。

该过程的流程文档如图 16-4 所示。



图 16-4 商品采购系统流程图

由于流程实例的运转依赖节点的出线,换言之如果节点没有定义出线,那么流程实例运转势必出错。上述流程分为 4 个阶段,对于供应商上新来说不需要进行回退操作,该节点可以退回的连线为 0,商务审核可以回退给供应商上新,该节点可以退回的连线为 1,运营审核可以回退给商务审核或者供应商上新,该节点可以退回的连线为 2,合同签订可以回退给运营审核、商务审核、供应商上新,该节点可以退回的连线为 3。

看到此处定义的回退连线可知,如果在合同签订之后增加一个审核部门,那么新增节点可以退回的连线为 4,以此类推。

如果流程文档中需要的回退操作不多,可以为节点定义不同的出线信息,如果流程中存在大量的节点以及不确定的回退操作,那么很显然为每个节点都定义一系列的出线不是明智选择。

那么有没有一种更好的方式,解决上面的问题呢?这也是接下来需要重点讲解的。

在实现节点跳转功能之前,需要考虑如下几个问题。

- (1) 确认需要跳转的节点以及跳转的目标节点。
- (2) 确认需要跳转节点归属的流程文档。
- (3) 获取需要跳转节点的流程实例与执行实例。
- (4) 确认跳转到目标节点之后是否需要添加变量?

(5) 确认跳转到目标节点之后是否需要触发目标节点配置的监听器?

(6) 如何处理流程实例存在分支或者多实例的情况?

16.4.1 常规节点跳转

首先以图 16-4 对应的流程文档为例,详细讲解常规节点(流程实例不存在分支的情况)的跳转操作,该流程文档的内容如代码清单 16-14 所示。

代码清单 16-14 common.bpmn

```
1 <process id="common" name="common" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="usertask1" name="供应商上新"></userTask>
4   <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
5   <userTask id="usertask2" name="商务审核"></userTask>
6   <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"></sequenceFlow>
7   <userTask id="usertask3" name="运营审核"></userTask>
8   <userTask id="usertask4" name="合同签订"></userTask>
9   <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="usertask3"></sequenceFlow>
10  <sequenceFlow id="flow4" sourceRef="usertask3" targetRef="usertask4"></sequenceFlow>
11  <endEvent id="endevent1" name="End"></endEvent>
12  <sequenceFlow id="flow5" sourceRef="usertask4" targetRef="endevent1"></sequenceFlow>
13 </process>
```

流程定义完毕将其部署并启动流程实例。ACT_RU_TASK 表新增的数据如图 16-5 所示。

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	NAME_
2505		1 2501	2501	common:14	供应商上新

图 16-5 ACT_RU_TASK 表新增的数据

接下来,定义一个命令类如代码清单 16-15 所示。

代码清单 16-15 ShareniuCommonJumpTaskCmd.java

```
1 public class ShareniuCommonJumpTaskCmd implements Command<Void> {
2   protected String executionId;          // 执行实例 id
3   protected String parentId;             // 流程实例 id
4   protected ActivityImpl desActivity;     // 目标节点
5   protected Map<String, Object> paramvar; // 变量
6   protected ActivityImpl currentActivity; // 当前的节点
7   public Void execute(CommandContext commandContext) {
8     // 获取当前流程的 executionId, 因为常规流程的执行实例 id 与流程实例 id 是相等的
9     ExecutionEntityManager executionEntityManager = Context
10       .getCommandContext().getExecutionEntityManager();
11     ExecutionEntity executionEntity = executionEntityManager.findExecutionById(parentId);
12     executionEntity.setVariables(paramvar);
13     executionEntity.setExecutions(null);
14     executionEntity.setEventSource(this.currentActivity);
```



```

15     executionEntity.setActivity(this.currentActivity);
16     //获取 TaskEntity 集合
17     Iterator<TaskEntity> localIterator = Context.getCommandContext()
18     .getTaskEntityManager().findTasksByProcessInstanceId(parentId).iterator();
19     while (localIterator.hasNext()) {
20         TaskEntity taskEntity = (TaskEntity) localIterator.next();
21         //触发任务监听器
22         taskEntity.fireEvent("complete");
23         //删除任务的原因
24         Context.getCommandContext().getTaskEntityManager()
25         .deleteTask(taskEntity, "shareniuCompleted", false);
26     }
27     //推动流程实例继续向下运转
28     executionEntity.executeActivity(this.desActivity);
29     return null;
30 }
31 }

```

对于常规节点的跳转来说,由于不存在分支情况,流程实例的 id 值与执行实例的 id 值是相等的。

上述代码中,第 9~10 行获取 ExecutionEntityManager 实例对象,第 11 行根据 parentId 获取 ExecutionEntity 实例对象,第 12~15 行为 executionEntity 对象填充属性,第 17~18 行获取流程实例 id 值为 parentId 下的所有任务节点(可以结合具体业务实现任务节点的查询工作),第 24~25 行删除任务节点,第 28 行设置流程实例需要运转到的目标节点,即 desActivity 值。

命令类定义完毕,需要调用该命令类完成节点的跳转操作,如代码清单 16-16 所示。

代码清单 16-16 App.java

```

1 public void testShareniuCommonJumpTaskCmd() throws IOException {
2     ReadonlyProcessDefinition processDefinitionEntity = ( ReadonlyProcessDefinition )
repositoryService
3     .getProcessDefinition("common:1:4");
4     //目标节点
5     ActivityImpl destinationActivity = (ActivityImpl)
6     processDefinitionEntity.findActivity("usertask4");
7     String executionId = "2501";
8     String parentId = "2501";
9     //当前节点
10    ActivityImpl currentActivity = (ActivityImpl) processDefinitionEntity.findActivity
("usertask1");
11    processEngine.getManagementService().executeCommand(
12    new ShareniuCommonJumpTaskCmd(executionId, parentId, destinationActivity, null,
13    currentActivity));
14 }

```

第2~3行获取 ReadOnlyProcessDefinition 实例对象,第5~6行获取目标节点,本案例目标节点 id 值为 usertask4 的任务节点,该节点的名称为合同签订,第7~8行定义了流程实例的 id 值,第10行获取当前的节点,即 id 为 usertask1 的任务节点,第11~13行执行 ShareniuCommonJumpTaskCmd 命令类。执行上述代码,ACT_RU_TASK 表新增的数据如图 16-6 所示。

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	NAME_
5002	1	2501	2501	common:1:4	usertask4	合同签订

图 16-6 ACT_RU_TASK 表新增的数据

根据图 16-6 可知流程实例已经成功运转到 id 为 usertask4 的任务节点了。

16.4.2 分支节点跳转

首先定义一个带有并行网关的流程文档,该流程文档的内容如代码清单 16-17 所示,流程图如图 16-7 所示。

代码清单 16-17 parallel.bpmn

```
1 <process id="parallel" name="parallel" isExecutable="true">
2   <startEvent id="startevent1" name="Start"/>
3   <userTask id="usertask1" name="usertask1"/>
4   <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"/>
5   <parallelGateway id="parallelgateway1" name="Parallel Gateway"/>
6   <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="parallelgateway1"/>
7   <userTask id="usertask2" name="usertask2"/>
8   <sequenceFlow id="flow3" sourceRef="parallelgateway1" targetRef="usertask2"/>
9   <userTask id="usertask3" name="usertask3"/>
10  <sequenceFlow id="flow4" sourceRef="parallelgateway1" targetRef="usertask3"/>
11  <parallelGateway id="parallelgateway2" name="Parallel Gateway"/>
12  <sequenceFlow id="flow5" sourceRef="usertask2" targetRef="parallelgateway2"/>
13  <sequenceFlow id="flow6" sourceRef="usertask3" targetRef="parallelgateway2"/>
14  <userTask id="usertask4" name="usertask4"/>
15  <sequenceFlow id="flow7" sourceRef="parallelgateway2" targetRef="usertask4"/>
16  <endEvent id="endevent1" name="End"/>
17  <sequenceFlow id="flow8" sourceRef="usertask4" targetRef="endevent1"/>
18 </process>
```

部署上述流程文档并启动流程实例,流程实例直接到达 usertask1 任务节点,然后调用相应的 API 完成该任务,这样流程实例会流转到 usertask2 和 usertask3 任务节点,ACT_RU_EXECUTION 表的数据变化如图 16-8 所示。

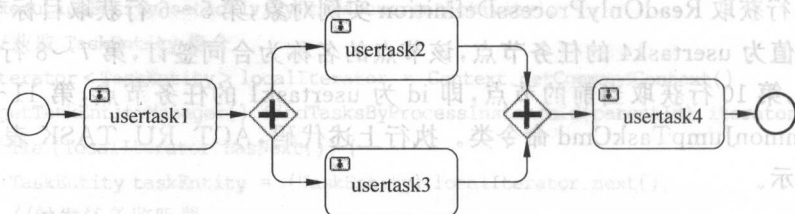


图 16-7 流程图

ID_	REV_	PROC_INST_ID_	ACT_ID_	PARENT_ID_	PROC_DEF_ID_
2501	2	2501	parallelgatewa	(Null)	parallel:1:4
5002	1	2501	usertask2	2501	parallel:1:4
5003	1	2501	usertask3	2501	parallel:1:4

图 16-8 ACT_RU_EXECUTION 表的数据变化

通过图 16-8 可知, ID_ 值为 5002 和 5003 的数据, PARENT_ID_ 值均为 2501, 而 ID_ 值为 2501 的记录对应流程实例的数据。

这个时候如果使用常规节点跳转方式将 usertask2 节点跳到 usertask1 任务节点, 是可以成功的, 但是会衍生一个问题, 跳转操作完毕再去完成 usertask3 任务节点, 程序就会报错。

扩展

并行网关的行为类为 ParallelGatewayActivityBehavior。

基于上述的讲解要想实现分支节点的跳转必须考虑如下几种情况。

(1) 当 usertask2 或者 usertask3 任意一个节点跳转到 usertask1 时, 必须要将存活于 ACT_RU_EXECUTION 和 ACT_RU_TASK 表中的另一个节点数据删除。

(2) 设置变量是流程实例级别的, 而不是分支级别的。

接下来, 定义一个命令类如代码清单 16-18 所示。

代码清单 16-18 ShareniuParallelJumpTaskCmd.java

```
1 public class ShareniuParallelJumpTaskCmd implements Command<Void> {
2     ...//属性定义参考 ShareniuCommonJumpTaskCmd 类
3     public Void execute(CommandContext commandContext) {
4         ExecutionEntityManager executionEntityManager = Context
5             .getCommandContext().getExecutionEntityManager();
6         ExecutionEntity executionEntity = executionEntityManager.findExecutionById(executionId);
7         String id = null;
8         if (executionEntity.getParent() != null) {
9             executionEntity = executionEntity.getParent();
10            if (executionEntity.getParent() != null) {
11                executionEntity = executionEntity.getParent();
12                id = executionEntity.getId();
13            } else {
14                id = executionEntity.getId();
15            }
16        }
```

```

16     }
17     executionEntity.setVariables(paramvar);
18     executionEntity.setEventSource(this.currentActivity);
19     executionEntity.setActivity(this.currentActivity);
20     // 根据 executionId 获取 TaskEntity
21     Iterator<TaskEntity> localIterator = Context.getCommandContext().getTaskEntityManager().
22     .getTaskEntityManager().findTasksByExecutionId(this.executionId).iterator();
23     while (localIterator.hasNext()) {
24         TaskEntity taskEntity = (TaskEntity) localIterator.next();
25         // 触发任务监听
26         taskEntity.fireEvent("complete");
27         // 删除任务的原因
28         Context.getCommandContext().getTaskEntityManager().
29         .deleteTask(taskEntity, "shareniuCompleted", false);
30     }
31     List<ExecutionEntity> list = executionEntityManager
32     .findChildExecutionsByParentExecutionId(parentId);
33     for (ExecutionEntity ee : list) {
34         ee.remove();
35     }
36     executionEntity.executeActivity(this.desActivity);
37     return null;
38 }
39 }

```

第6行获取 ExecutionEntity 实例对象,第8~16行如果当前的执行实例对象存在流程实例对象,则获取流程实例对象。

第31~35行用于删除执行实例的数据,该步骤非常的重要。

上述步骤执行完毕,开始调用该命令类,其中目标节点为 usertask1。操作完毕 ACT_RU_TASK 表新增数据如图 16-9 所示。

ID_	REV_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	NAME_
7502	1	2501	2501	parallel:14	usertask1	usertask1

图 16-9 ACT_RU_TASK 表新增的数据

分支节点的跳转功能已经实现并验证通过,但是也存在一个小瑕疵,ACT_HI_PROCINST 表中的 END_TIME_ 与 DURATION_ 字段并没有被更新,带着这个问题学习多实例节点功能的实现。

16.4.3 多实例节点跳转

首先定义一个带有多实例任务节点的流程文档,该流程文档的内容如代码清单 16-19 所示,流程图如图 16-10 所示。

代码清单 16-19 multiInstance.bpmn

```

1 <process id="multiInstance" name="multiInstance" isExecutable="true">
2 <startEvent id="startevent1" name="Start"></startEvent>

```



```
3 <userTask id="usertask1" name="多实例任务"
4   activiti:candidateUsers="shareniu1,shareniu2,shareniu3">
5   <multiInstanceLoopCharacteristicsisSequential="false">
6   <loopCardinality>3</loopCardinality>
7   </multiInstanceLoopCharacteristics>
8 </userTask>
9 <endEvent id="endevent1" name="End"></endEvent>
10 <userTask id="usertask2" name="usertask2"></userTask>
11 <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"></sequenceFlow>
12 <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>
13 <sequenceFlow id="flow4" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
14 </process>
```

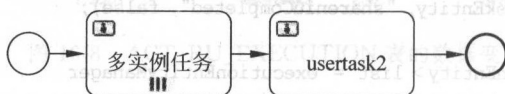


图 16-10 流程图

在上述代码中,第3~8行定义了id为usertask1的任务节点,其中第5行定义了多实例按照并行方式执行,第6行定义了该任务节点可以执行三次。

部署上述的流程文档并启动流程实例,这样流程实例直接到达usertask1任务节点,ACT_RU_EXECUTION表的数据变化如图16-11所示。

ID_	REV_	PROC_INST_ID_	ACT_ID_	PARENT_ID_	PROC_DEF_ID_
10001	1	10001	{Null}	{Null}	multiinstance:1:7504
10003	1	10001	usertask1	10001	multiinstance:1:7504
10008	1	10001	usertask1	10003	multiinstance:1:7504
10009	1	10001	usertask1	10003	multiinstance:1:7504
10010	1	10001	usertask1	10003	multiinstance:1:7504

图 16-11 ACT_RU_EXECUTION表的数据变化

通过图16-11可知,对于多实例任务节点而言,ID_值为10008、10009、10010均为执行实例数据,其中PARENT_ID_值为10003,ID_值为10003的PARENT_ID_值为10001。

注意

多实例任务节点存在流程实例和两个执行实例,因此处理的时候有别于分支节点跳转。

接下来,定义一个命令类如代码清单16-20所示。

代码清单 16-20 SharenjuParallelJumpTaskCmd.java

```
1 public class SharenjuMultiInstanceJumpTaskCmd implements Command<Void> {
2   ...//属性定义参考 SharenjuCommonJumpTaskCmd 类
3   public Void execute(CommandContext commandContext) {
4     ExecutionEntityManager executionEntityManager = Context
5     .getCommandContext().getExecutionEntityManager();
6     ExecutionEntity executionEntity = executionEntityManager.findExecutionById(executionId);
```

```

7      String id = null;
8      if (executionEntity.getParent() != null) {
9          executionEntity = executionEntity.getParent();
10     if (executionEntity.getParent() != null) {
11         executionEntity = executionEntity.getParent();
12         id = executionEntity.getId();
13     }
14     id = executionEntity.getId();
15 }
16 executionEntity.setVariables(paramvar);
17 executionEntity.setExecutions(null);
18 executionEntity.setEventSource(this.currentActivity);
19 executionEntity.setActivity(this.currentActivity);
20 // 根据 executionId 获取 Task
21 Iterator<TaskEntity> localIterator = Context.getCommandContext()
22     .getTaskEntityManager().findTasksByProcessInstanceId(id).iterator();
23 while (localIterator.hasNext()) {
24     TaskEntity taskEntity = (TaskEntity) localIterator.next();
25     // 触发任务监听
26     taskEntity.fireEvent("complete");
27     // 删除任务的原因
28     Context.getCommandContext().getTaskEntityManager().deleteTask(taskEntity, "completed",
29 false);
30 }
31 List<ExecutionEntity> list = executionEntityManager
32     .findChildExecutionsByParentExecutionId(parentId);
33 for (ExecutionEntity executionEntity2 : list) {
34     ExecutionEntity findExecutionById = executionEntityManager.findExecutionById(executionEntity2.getId());
35     List<ExecutionEntity> parent = executionEntityManager
36         .findChildExecutionsByParentExecutionId(executionEntity2.getId());
37     for (ExecutionEntity executionEntity3 : parent) {
38         executionEntity3.remove();
39         Context.getCommandContext().getHistoryManager().recordActivityEnd(executionEntity3);
40     }
41     executionEntity2.remove();
42     Context.getCommandContext().getHistoryManager().recordActivityEnd(executionEntity2);
43 }
44 commandContext.getIdentityLinkEntityManager().deleteIdentityLinksByProcInstance(id);
45 executionEntity.executeActivity(this.desActivity);
46 return null;
47 }
48 }

```

第 7~14 行查找执行实例对应的顶级流程实例，直到找到为止，第 20~22 行查找顶级流程实例下的所有任务，然后第 23~29 行循环遍历任务并将其删除，第 30~31 行获取流程实例下的执行实例，对应 ID_值为 10003 的数据，第 34~35 行遍历 ID_值为 10003 下的所有执行实例集合，第 36~39 行删除执行实例，对应 ID_值为 10008、10009、10010 的数据。

第 40 行删除 ID_值为 10003 的数据，并在第 41 行将其更新到 ACT_HI_ACTINST

表中。

第 43 行删除当前流程实例对应的 ACT_RU_IDENTITYLINK 表中的数据,以避免脏数据。

由于外键约束的原因,上述的代码必须按照顺序执行,不能调换位置。

调用该命令类,其中目标节点为 usertask2。操作完毕 ACT_RU_TASK 表新增的数据如图 16-12 所示。

ID	REV	PROC_INST_ID	ACT_ID	PROC_DEF_ID
10001	2	10001	usertask2	multiInstance:1:7504

图 16-12 ACT_RU_TASK 表新增的数据

16.5 会签

在实际项目开发中有这样的一个例子,假如一个部门内部的投票,该部门有 5 个人,那么当这 5 个人都进行投票的时候可以分为如下几种情况。

(1) 该部门所有的人都需要投票,当所有人投票完毕,则当前节点结束,路程实例继续向下运转(人人参与)。

(2) 该部门所有的人都需要投票,只要有一半以上的人同意,则当前节点结束,路程实例继续向下运转(部分人参与)。

(3) 该部门只要部门经理投票通过,则当前节点结束,流程实例继续向下运转(一票否决权)。

(4) 该部门入职一位新人,新人也需要投票,所有人投票完毕,则当前节点结束,路程实例继续向下运转(人员增加)。

(5) 该部门离职一位员工,离职员工不需要投票,其余所有人员投票完毕,则当前节点结束,路程实例继续向下运转(人员减少)。

(6) 该部门有一员工出差,让其他员工代替投票,则当前节点结束,路程实例继续向下运转(人员不变,但节点处理人需要变化)。

(7) 该部门中职位不同投票的权重也不相同,例如 5 个人员权重加起来为 1,部门经理的权重为 50%,组长的权重 20%,普通员工的权重为 10%。然后根据权重计算投票结果(权重不同)。

(8) 该部门所有人依次开始投票,普通员工一个个投票,然后组长投票,最后部门经理投票并统计结果(串行执行)。

扩展

并行多实例的行为类为 ParallelMultiInstanceBehavior,串行多实例的行为类为 SequentialMultiInstanceBehavior。

16.5.1 串行多实例

首先讲解串行多实例的使用,定义一个流程文档如代码清单 16-21 所示。

代码清单 16-21 sequential.bpmn

```
1 <process id="multiInstance" name="multiInstance" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="usertask1" name="多实例任务" activiti:candidateUsers="sharenui1,
sharenui2">
4     <multiInstanceLoopCharacteristics isSequential="false">
5       <loopCardinality>2</loopCardinality>
6     </multiInstanceLoopCharacteristics>
7   </userTask>
8   <endEvent id="endevent1" name="End"></endEvent>
9   <userTask id="usertask2" name="usertask2"></userTask>
10  <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"></sequenceFlow>
11  <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>
12  <sequenceFlow id="flow4" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
13 </process>
```

在上述代码中,第3~7行定义了id为usertask1的任务节点,其中activiti:candidateUsers属性表示可以处理当前任务的候选人集合,如果存在多个候选人,使用“,”进行分割。

第4行中的isSequential属性:true表示当前任务节点并行执行,false表示串行执行。对于该案例而言,如果该值为true,则sharenui1和sharenui2可以同时处理该任务;如果该值为false,则sharenui1处理完毕,sharenui2才可以处理该任务。

第5行中loopCardinality元素中配置的值为2,表示当前任务循环2次结束。

将上述流程文档部署并启动流程实例,ACT_RU_VARIABLE表的变化如图16-13所示。

ID_	REV_	TYPE_	NAME_	EXECUTION_ID_	PROC_INST_ID_	TEXT_
2505		1 integer	nrOfInstances	2503	2501	2
2506		1 integer	nrOfCompletedInstances	2503	2501	0
2507		1 integer	loopCounter	2503	2501	0
2508		1 integer	nrOfActiveInstances	2503	2501	1

图 16-13 ACT_RU_VARIABLE 表新增的数据

对上图中的涉及的变量说明如下。

(1) nrOfInstances: 实例的数量。

(2) nrOfCompletedInstances: 已经完成的实例个数。

(3) nrOfActiveInstances: 当前还没有完成的实例个数,如果使用串行的方式实现多实例则该值永远为1。

(4) loopCounter: 循环次数,串行方式计算多实例任务是否可以离开当前节点依赖该值,并行方式不依赖该值。对于串行方式而言,该值从0开始,当多实例中的任意一个任务完成之后,该值加1。该变量的操作可以参考SequentialMultiInstanceBehavior类中的leave方法。

了解了以上变量的含义之后,查看ACT_RU_TASK表中多实例任务中的ASSIGNEE_列,该列为空并没有数据,因为候选人或者候选组的信息存放在ACT_RU_IDENTITYLINK表中,因此要想完成该任务节点,首先需要认领任务,所以该方式不灵活。ACT_RU_TASK

表新增数据如图 16-14 所示。

ID_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	ASSIGNEE_	NAME
2509	2503	2501	sequential:14	usertask1	(Null)	多实例任务

图 16-14 ACT_RU_TASK 表新增的数据

16.5.2 认领和归还任务

对于任务的认领操作而言需要调用 TaskService 中的 claim 方法,具体实现如代码清单 16-22 所示。

代码清单 16-22 App.Java

```
1 public void testclaim() throws IOException {
2     taskService.claim("2509", "shareniu1");
3 }
```

上述代码执行完毕,ACT_RU_TASK 表中的数据变化如图 16-15 所示。

ID_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	ASSIGNEE_	NAME
2509	2503	2501	sequential:14	usertask1	shareniu1	多实例任务

图 16-15 ACT_RU_TASK 表新增的数据

将上述代码中的 shareniu1 修改为 shareniu2 并再次执行,程序抛出 ActivitiTaskAlreadyClaimedException 异常,因为已经被认领的任务是不能被再次认领的。

也可以通过 TaskService 中的 setAssignee 方法修改 ACT_RU_TASK 表中的 ASSIGNEE_ 值,该方法不会检查任务是否被拾取。

归还任务就是将 ACT_RU_TASK 表中的 ASSIGNEE_ 值设置为空,如代码清单 16-23 所示。

代码清单 16-23 App.Java

```
1 public void testunclaim() throws IOException {
2     taskService.unclaim("2509");
3 }
```

16.5.3 代理任务

以上文的投票案例为例,如果 shareniu1 可以处理任务节点,但是 shareniu1 现在出差没法办理该任务,需要将任务委托给 shareniu2 处理,对于该需求来说,shareniu1 为任务的归属者,shareniu2 为任务的办理人,具体实现如代码清单 16-24 所示。

代码清单 16-24 App.Java

```
1 public void testAgent() throws IOException {
```

```
2 String taskId = "2509";
3 taskService.setOwner(taskId, "shareniu1");
4 taskService.setAssignee(taskId, "shareniu2");
5 }
```

执行上述代码,ACT_RU_TASK 表中的数据变化如图 16-16 所示。

ID_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	OWNER_	ASSIGNEE_
2509	2503	2501	sequential:1:4	usertask1	shareniu1	shareniu2

图 16-16 ACT_RU_TASK 表新增的数据

对于 OWNER_列,仅仅是记录任务的归属者,例如同一个任务被反复代理,但是 Activiti 并没有记录代理的操作痕迹。实际项目开发中可以扩展一张表用于记录代理的详细信息。

16.5.4 并行多实例

相对而言并行多实例的应用场景更多,首先定义一个流程文档如代码清单 16-25 所示。

代码清单 16-25 collectionmultiInstance.bpmn

```
1 <process id="collectionmultiInstance" name="collectionmultiInstance" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="usertask1" name="多实例任务" activiti:assignee="${shareniuAssignee}">
4     <multiInstanceLoopCharacteristics isSequential="false" activiti:collection="
      assigneeList"
5     activiti:elementVariable="shareniuAssignee">
6     <completionCondition>${nrOfCompletedInstances/nrOfInstances}>= 0.75}
7     </completionCondition>
8     </multiInstanceLoopCharacteristics>
9   </userTask>
10  <endEvent id="endevent1" name="End"></endEvent>
11  <userTask id="usertask2" name="usertask2"></userTask>
12  <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"></sequenceFlow>
13  <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>
14  <sequenceFlow id="flow4" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
15 </process>
```

在上述代码中,第 3~9 行定义了 id 为 usertask1 的任务节点,其中 activiti:assignee 属性表示任务的处理人,该值与第 5 行中的 activiti:elementVariable 配合起来使用。第 4 行中的 activiti:collection 属性为集合的名称,引擎会根据集合中元素的个数生成对应的执行实例(生成执行实例的个数与集合元素的个数一致),例如集合中元素的个数为 5,则表示当前的任务会循环 5 次。流程实例到达该任务节点之前需要设置 assigneeList 变量。

第 6 行中的 completionCondition 表示当前任务通过的条件,对于该配置而言,例如现在在有 4 个人需要投票,则其中任意 3 个人投票之后,当前的任务节点结束,如果不配置该值,默认情况下通过条件为 1(需要 4 个人投票)。

将上述流程文档部署并启动流程实例,其中启动流程实例的方式如代码清单 16-26 所示。

代码清单 16-26 App.Java

```
1 public void startProcessInstanceById() throws IOException {
2     Map<String, Object> vars = new HashMap<String, Object>();
3     String[] v = {"shareniu1", "shareniu2", "shareniu3"};
4     vars.put("assigneeList", Arrays.asList(v));
5     runtimeService.startProcessInstanceById("collectionmultiInstance:1:4", vars);
6 }
```

上述代码执行完毕,ACT_RU_TASK 表中的数据变化如图 16-17 所示。

ID_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	ASSIGNEE_
2516	2511	2501	collectionmultiInstance:1:4	usertask1	shareniu1
2521	2512	2501	collectionmultiInstance:1:4	usertask1	shareniu2
2526	2513	2501	collectionmultiInstance:1:4	usertask1	shareniu3

图 16-17 ACT_RU_TASK 表新增的数据

使用集合方式操作多实例节点,引擎会将集合中的元素自动作为任务处理人。

16.5.5 加签

对于上述并行多实例案例而言,如果需要 id 为 usertask1 的任务节点添加一个处理人 shareniu4,如何实现呢?思路如下。

(1) 需要在 ACT_RU_TASK 表中增加一条数据,新增的数据与图 16-17 中的数据相似,不同的是 ASSIGNEE_列和 ID_列的值。其中,ASSIGNEE_值为 shareniu4,ID_值可以使用 id 生成器生成。如何获取新增数据 EXECUTION_ID_和 PROC_INST_ID_列的值?这两个列均为外键约束,对应 ACT_RU_EXECUTION 表中的 ID_列,由于外键约束的缘故,必须首先在 ACT_RU_EXECUTION 表插入一条数据。

(2) 既然新增处理人,则 ACT_RU_VARIABLE 表中的 nrOfInstances,nrOfActiveInstances 变量值需要在原有的基础之上加一。

接下来就按照上述的思路开始设计,首先定义一个命令类如代码清单 16-27 所示。

代码清单 16-27 ShareniuCountersignAddCmd.Java

```
1 public class ShareniuCountersignAddCmd implements Command<Void>{
2     protected String executionId; //执行实例
3     protected String assignee; //执行实例
4     public Void execute(CommandContext commandContext) {
5         ProcessEngineConfigurationImpl pec = commandContext.getProcessEngineConfiguration();
6         RuntimeService runtimeService = pec.getRuntimeService();
7         TaskService taskService = pec.getTaskService();
8         IdGenerator idGenerator = pec.getIdGenerator();//获取 id 生成器
9         //获取 Execution 实例对象
10        Execution execution = runtimeService.createExecutionQuery().executionId(executionId).single-
```

```

Result();
11  ExecutionEntity ee = (ExecutionEntity) execution;
12  ExecutionEntity parent = ee.getParent(); //获取父级 ExecutionEntity 实例对象
13  ExecutionEntity newExecution = parent.createExecution();
                                           //创建 ExecutionEntity 实例对象
14  newExecution.setActive(true); //设置为激活状态
15  //该属性表示创建的新Execution对象为分支,非常重要,不可缺少
16  newExecution.setConcurrent(true);
17  newExecution.setScope(false);
18  Task newTask = taskService.createTaskQuery().executionId(executionId).singleResult();
19  TaskEntity t = (TaskEntity) newTask;
20  TaskEntity taskEntity = new TaskEntity();
21  taskEntity.setCreateTime(new Date());
22  taskEntity.setTaskDefinition(t.getTaskDefinition());
23  taskEntity.setProcessDefinitionId(t.getProcessDefinitionId());
24  taskEntity.setTaskDefinitionKey(t.getTaskDefinitionKey());
25  taskEntity.setProcessInstanceId(t.getProcessInstanceId());
26  taskEntity.setExecutionId(newExecution.getId());
27  taskEntity.setName(newTask.getName());
28  String taskId = idGenerator.getNextId();
29  taskEntity.setId(taskId);
30  taskEntity.setExecution(newExecution);
31  taskEntity.setAssignee(assignee);
32  taskService.saveTask(taskEntity);
33  int loopCounter = SharenuiLoopVariableUtils.getLoopVariable(newExecution, "nrOfInstances");
34  int nrOfCompletedInstances = SharenuiLoopVariableUtils.getLoopVariable(newExecution,
35  "nrOfActiveInstances");
36  SharenuiLoopVariableUtils.setLoopVariable(newExecution, "nrOfInstances", loopCounter + 1);
37  SharenuiLoopVariableUtils.setLoopVariable(newExecution, "nrOfActiveInstances", nrOfC-
38  ompletedInstances + 1);
39  return null;
40 }

```

第2行定义了 executionId,该值可以为图 16-17 中任意一个 EXECUTION_ID_列的值,第3行定义了 assignee,该值为新的任务处理人,第8行获取 id 生成器,第10~11行通过 executionId 获取 ExecutionEntity 实例对象,然后第12行通过该实例对象获取其 parent,第13行通过 parent 创建 newExecution,然后在第14~17行为其填充属性,该操作非常重要,直接影响新增数据是否可用。

第18~32行创建 newTask 并将其入库,其中第26行和第30行将其与 newExecution 进行关联。

第33~37行获取和更新 ACT_RU_VARIABLE 表中的 nrOfInstances,nrOfActiveInstances 变量值,由于该操作比较通用,因此将通用方法放置到 SharenuiLoopVariableUtils 类中,该类中的核心方法如代码清单 16-28 所示。

代码清单 16-28 SharenuiLoopVariableUtils. Java

```

1 public class SharenuiLoopVariableUtils {

```



```
2 public static void setLoopVariable(ExecutionEntity execution,  
3 String variableName, Object value) {  
4     ActivityExecution parent = execution.getParent(); // 获取执行实例的父级  
5     parent.setVariableLocal(variableName, value); // 设置变量  
6 }  
7 public static Integer getLoopVariable(ExecutionEntity execution, String variableName) {  
8     Object value = execution.getVariableLocal(variableName); // 获取变量  
9     ActivityExecution parent = execution.getParent();  
10    while (value == null && parent != null) {  
11        value = parent.getVariableLocal(variableName);  
12        parent = parent.getParent();  
13    }  
14    return (Integer) (value != null ? value : 0);  
15 }
```

上述工作完毕,新建一个测试类,如代码清单 16-29 所示。

代码清单 16-29 App. Java

```
1 public void testSharenIUCountersignAddCmd() throws IOException {  
2     processEngine.getManagementService().executeCommand(  
3         new SharenIUCountersignAddCmd("7513", "sharenIU4"));  
4 }
```

执行上述代码,ACT_RU_TASK 表中的数据变化如图 16-18 所示。ACT_RU_VARIABLE 表中的数据变化如图 16-19 所示。

ID	EXECUTION_ID	PROC_INST_ID	PROC_DEF_ID	TASK_DEF_KEY	ASSIGNEE
10002	10001	7501	collectionmultiInstance:1:4	usertask1	sharenIU4
7516	7511	7501	collectionmultiInstance:1:4	usertask1	sharenIU1
7521	7512	7501	collectionmultiInstance:1:4	usertask1	sharenIU2
7526	7513	7501	collectionmultiInstance:1:4	usertask1	sharenIU3

图 16-18 ACT_RU_TASK 表新增的数据

ID	REV	TYPE	NAME	EXECUTION_ID	PROC_INST_ID	TEXT
7508	2 integer	nrOfInstances	7506	7501	4	
7509	1 integer	nrOfCompletedInstances	7506	7501	0	
7510	2 integer	nrOfActiveInstances	7506	7501	4	

图 16-19 ACT_RU_VARIABLE 表新增的数据

至此加签功能已经实现完毕。

16.5.6 减签和退签

上面讲解了如何实现加签功能,接下来讲解减签功能的实现,减签就是将指定的任务数删除即可,操作的表有 ACT_RU_TASK、ACT_RU_EXECUTION、ACT_RU_VARIABLE。

定义一个实现减签功能的命令类如代码清单 16-30 所示。

代码清单 16-30 ShareniuCountersignMinCmd. Java

```

1 public class ShareniuCountersignMinCmd implements Command<Void> {
2     protected String taskId; //任务 id
3     ...//省略构造方法
4     public Void execute(CommandContext commandContext) {
5         ProcessEngineConfigurationImpl pec = commandContext.getProcessEngineConfiguration();
6         TaskService taskService = pec.getTaskService();
7         RuntimeService runtimeService = pec.getRuntimeService();
8         TaskEntity task = (TaskEntity) taskService.createTaskQuery().taskId(taskId).singleResult();
9         String executionId = task.getExecutionId(); // 获取 executionId
10        ExecutionEntity execution = (ExecutionEntity) runtimeService
11            .createExecutionQuery().executionId(executionId).singleResult();
12        int loopCounter = ShareniuLoopVariableUtils.getLoopVariable(execution, "nrOfInstances");
13        int nrOfCompletedInstances = ShareniuLoopVariableUtils.getLoopVariable(
14            execution, "nrOfActiveInstances");
15        ShareniuLoopVariableUtils.setLoopVariable(execution, "nrOfInstances", loopCounter - 1);
16        ShareniuLoopVariableUtils.setLoopVariable(execution,
17            "nrOfActiveInstances", nrOfCompletedInstances - 1);
18        task.setProcessInstanceId(null);
19        task.setExecutionId(null);
20        taskService.saveTask(task);
21        ExecutionEntityManager executionEntityManager = commandContext.getExecutionEntityManager();
22        executionEntityManager.deleteProcessInstance(executionId, "shareniu", false);
23        taskService.deleteTask(taskId, false);
24        return null;
25    }
26 }

```

在上述代码中,第8行根据 taskId 获取 TaskEntity 实例对象,第12~17行分别获取并更新 nrOfInstances、nrOfActiveInstances,第18~19行分别设置 task 对象中的 processInstanceId、executionId 属性值为空,该操作非常重要,目的是为了解除 ACT_RU_TASK 表与 ACT_RU_EXECUTION 表数据之间的依赖,第20行将 task 对象的数据更新到会话缓存,第22行删除 ACT_RU_EXECUTION 表中的数据,第23行删除 ACT_RU_TASK 表中的数据。由于外键约束的缘故,以上操作步骤必须按顺序执行,不能调整。

新建一个测试类,如代码清单 16-31 所示。

代码清单 16-31 App. Java

```

1 public void testShareniuCountersignMinCmd(). {
2     processEngine.getManagementService().executeCommand(new ShareniuCountersignMinCmd("10002"));
3     processEngine.getManagementService().executeCommand(new ShareniuCountersignMinCmd("7516"));
4     processEngine.getManagementService().executeCommand(new ShareniuCountersignMinCmd("7521"));
5 }

```

以上操作的任务 id 值可以参考图 16-18。执行上述代码,ACT_RU_TASK 表中的数据变化如图 16-20 所示,ACT_RU_VARIABLE 表中的数据变化如图 16-21 所示。

ID_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_	ASSIGNEE_
7526	7513	7501	collectionmultiInstance:1:4	usertask1	shareniu3

图 16-20 ACT_RU_TASK 表的数据变化

ID_	REV_	TYPE_	NAME_	EXECUTION_ID_	PROC_INST_ID_	TEXT_
7508	5	integer	nrOfInstances	7506	7501	1
7509	1	integer	nrOfCompletedInstances	7506	7501	0
7510	5	integer	nrOfActiveInstances	7506	7501	1

图 16-21 ACT_RU_VARIABLE 表的数据变化

目前 ACT_RU_TASK 表中只有 ID_ 值为 7526 的数据,调用相应的 API 完成该任务,ACT_RU_TASK 表中的数据变化如图 16-22 所示。

ID_	EXECUTION_ID_	PROC_INST_ID_	PROC_DEF_ID_	TASK_DEF_KEY_
12502	7501	7501	collectionmultiInstance:1:4	usertask2

图 16-22 ACT_RU_TASK 表的数据变化

至此减签功能已经实现并测试通过,关于退签功能可以结合该案例自行实现。

16.6 会签节点自定义权重实现

16.6.1 定义处理人权重

首先定义一个流程文档如代码清单 16-32 所示,该流程文档图如图 16-23 所示。

代码清单 16-32 weight.bpmn

```

1 <process id="wegitInstance" name="wegitInstance" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="usertask1" name="多实例任务" activiti:assignee="{ ${shareniuAssignee} }">
4     <extensionElements>
5       <shareniu:shareniuOperations>
6         <shareniu:wegit>{"conditions":60,"info":{"shareniu1":80,"shareniu2":10,"shareniu3":10}}
7       </shareniu:wegit>
8     </shareniu:shareniuOperations>
9   </extensionElements>
10   <multiInstanceLoopCharacteristics isSequential="false" activiti:collection="
"assigneeList"
11     activiti:elementVariable="shareniuAssignee">
12     <completionCondition>{ ${shareniuX} }</completionCondition>
13   </multiInstanceLoopCharacteristics>
14 </userTask>
15 <endEvent id="endevent1" name="End"></endEvent>
16 <sequenceFlow id="flow4" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
17 <userTask id="usertask2" name="usertask2"></userTask>
18 <sequenceFlow id="flow5" name="通过" sourceRef="usertask1"
19   targetRef="exclusivegateway1"></sequenceFlow>
20 <userTask id="usertask3" name="usertask3"></userTask>

```

```
21 <sequenceFlow id="flow7" sourceRef="usertask3" targetRef="endevent1"></sequenceFlow>
22 <sequenceFlow id="flow8" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>
23 <exclusiveGateway id="exclusivegateway1" name="Exclusive Gateway"></exclusiveGateway>
24 <sequenceFlow id="flow9" sourceRef="exclusivegateway1" targetRef="usertask2">
25 <conditionExpression xsi:type="tFormalExpression"><![CDATA[ ${pass == true} ]]>
26 </conditionExpression>
27 </sequenceFlow>
28 <sequenceFlow id="flow10" name="不通过" sourceRef="exclusivegateway1" targetRef =
"usertask3">
29 <conditionExpression xsi:type="tFormalExpression">
30 <![CDATA[ ${pass == false} ]]></conditionExpression>
31 </sequenceFlow>
32 </process>
```

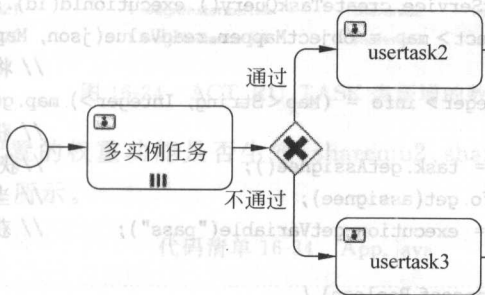


图 16-23 流程图

在上述代码中,第3~14行定义了id为usertask1的任务节点,其中第6行定义了任务处理人的权重信息,shareniu1的权重为80,shareniu2和shareniu3的权重都为10,conditions为60,表示当前任务通过的条件。

如果shareniu1处理该任务节点并投票为同意则权重为80,80很显然大于60,因此shareniu1处理当前的任务节点之后,当前的任务节点需要立即结束。如果shareniu1处理该任务节点并投票为不同意则权重为-80,-80小于60,因此shareniu2和shareniu3会继续处理当前节点。对于该案例而言,shareniu1、shareniu2、shareniu3权重相加为100。

第12行定义了当前节点的结束条件,需要引擎运行时计算,如果权重大于或者等于60,则需要设置"shareniuX"变量值为true,否则设置该值为false。

如果权重值大于或者等于60,则流程实例运转到id为usertask2的任务节点,否则运转到id为usertask3的任务节点。

16.6.2 获取权重信息并自动计算

由于并行多实例行为类并不能实现该功能,因此需要重写ParallelMultiInstanceBehavior类中的leave方法,在实现该功能之前先思考如下几个问题。

(1) 如何获取自定义权重信息。有了前面章节学习的基础,这个实现起来比较简单。

(2) 在完成多实例任务的同时,需要获取当前任务处理人对该任务的最终处理结果,该值可以通过变量的方式记录,例如同意则设置pass变量值为true,否则设置该变量值为false。

定义一个类并继承 ParallelMultiInstanceBehavior 类,该类的核心定义如代码清单 16-33 所示。

代码清单 16-33 ShareniuParallelMultiInstanceBehavior.java

```

1 public class ShareniuParallelMultiInstanceBehavior extends ParallelMultiInstanceBehavior{
2     ...//省略构造方法
3     public void leave(ActivityExecution execution) {
4         String json = "";
5         ...//省略权重信息的获取
6         String id = execution.getId();
7         ObjectMapper objectMapper = new ObjectMapper();
8         try {
9             TaskService taskService = execution.getEngineServices().getTaskService();
10            Task task = taskService.createTaskQuery().executionId(id).singleResult();
11            Map<String, Object> map = objectMapper.readValue(json, Map.class);
12
13            // 将 json 转化为 Map
14            Map<String, Integer> info = (Map<String, Integer>) map.get("info");
15
16            // 获取权重信息
17            String assignee = task.getAssignee(); //获取任务处理人
18            int weight = info.get(assignee); // 当前处理人的权重值
19            Object variable = execution.getVariable("pass"); // 获取变量
20            int x = 0;
21            if(variable instanceof Boolean) {
22                boolean flag = (boolean) variable;
23                if (flag) { //同意
24                    if (execution.getParent() != null) {
25                        Integer loopVariable = getLoopVariable(execution, "weight");
26                        //从数据库中获取 weight 变量
27                        x = weight + loopVariable;
28                        setLoopVariable(execution.getParent(), "weight", x); //设置 weight 变量
29                    }
30                } else { //不同意
31                    if (execution.getParent() != null) {
32                        Integer loopVariable = getLoopVariable(execution, "weight");
33                        x = loopVariable - weight;
34                        setLoopVariable(execution.getParent(), "weight", x);
35                    }
36                }
37            }
38            if (x >= (int) map.get("conditions")) { //获取通过的权重值
39                execution.setVariable("shareniuX", true); //表示多实例可以结束
40            } else {
41                execution.setVariable("shareniuX", false);
42            }
43            super.leave(execution);
44        } catch (Exception e) {}
45    }
46 }

```

第 5 行获取并解析扩展的权重信息,该过程可以参考第 14.4 节以及第 5.3 节,第 11 行将获取的权重数据格式(JSON)转化为 Map,第 17~32 行使用 weight 变量记录任务处理人的权重信息,并更新到数据库,第 33~37 行计算当前任务节点是否可以离开,如果可以则设置 shareniuX 变量值为 true,否则将其设置为 false,第 38 行调用父类的 leave 方法进行处理。

上述类定义完毕需要通过自定义活动工厂类将其注入流程引擎配置类,该过程可以参考第 14.4 节。

参考上文的讲解将该流程文档部署并启动新的流程实例,ACT_RU_TASK 表中的数据变化如图 16-24 所示。

ID_	REV_	PROC_DEF_ID_	ASSIGNEE	TASK_DEF_KEY_
22516	1	wegitInstance:1:4	shareniu1	usertask1
22521	1	wegitInstance:1:4	shareniu2	usertask1
22526	1	wegitInstance:1:4	shareniu3	usertask1

图 16-24 ACT_RU_TASK 表新增的数据

为了便于测试配置的权重功能是否生效,shareniu2、shareniu3、shareniu1 依次执行任务,如代码清单 16-34 所示。

代码清单 16-34 App.java

```
1 public void testComplete() throws IOException {
2     Map<String, Object> map = new HashMap<String, Object>();
3     map.put("pass", true);
4     taskService.complete("22521", map);
5     map.put("pass", true);
6     taskService.complete("22526", map);
7     map.put("pass", false);
8     taskService.complete("22516", map);
9 }
```

第 3~4 行 shareniu2 完成任务并投票为同意,第 5~6 行 shareniu3 完成任务并投票为同意,第 7~8 行 shareniu1 完成任务并投票为不同意。

执行上述代码,ACT_RU_TASK 表中的数据变化如图 16-25 所示。

ID_	REV_	PROC_DEF_ID_	TASK_DEF_KEY_	NAME_
25006	1	wegitInstance:1:4	usertask3	usertask3

图 16-25 ACT_RU_TASK 表新增的数据

当前的流程实例已经成功运转到 usertask3 任务节点,将上述代码中的第 7 行 pass 变量设置为 true,则当前的流程实例可以成功运转到 usertask2 任务节点。

16.6.3 优化建议

在上述案例中,自定义的权重信息记录在流程文档中,在实际项目开发中也可以将该权重信息存储在 Redis 或者数据库中,以方便开发人员修改,如果开发人员觉得权重不符合需

求,可以灵活修改,而无须再次部署流程文档并启动流程实例。

对于并行多实例而言,如果离开当前节点的条件满足,则该节点所有的出线都会执行,多实例节点并没有通过或者不通过的概念,因此上述案例实现过程中使用了排他网关,如果不打算使用排他网关,就需要扩展 `ParallelMultiInstanceBehavior` 类中的方法。

配置权重值时尽量使用 `integer` 类型,避免造成数据丢失。

设置变量时需要考虑流程实例级别的变量与执行实例级别的变量之间的区别。

16.7 接管 Activiti

在实际项目开发中,只要使用到工作流,那么代办、已办、转办等操作是必不可少的一环,而该过程就涉及了处理人的概念。例如,当前的任务节点处理人是 `shareniu1`,则 `shareniu1` 查询代办任务时可以把其需要处理的任务列表查询出来,如果 `shareniu1` 已经办理了一些任务,相应的也可以通过查询已办将已经办理的任务列表查询出来。在 Activiti 中,均可以通过 `TaskService` 接口提供的 API 操作任务,如果这一系列的 API 不能满足项目开发需求,那么就需要考虑如何扩展这些功能。

Activiti 中可以通过候选组或者候选人的方式为任务指定办理人,该操作涉及的表有 `ACT_RU_TASK`、`ACT_RU_IDENTITYLINK`、`ACT_HI_IDENTITYLINK`。`ACT_RU_TASK` 表存储与任务有关的信息,`ACT_RU_IDENTITYLINK` 表存储候选组或者候选人的信息,`ACT_HI_IDENTITYLINK` 表主要用于对候选人或者候选组的信息进行备案记录。

默认情况下用户与组需要操作的表有 `ACT_ID_USER`、`ACT_ID_GROUP`、`ACT_ID_MEMBERSHIP`。`ACT_ID_USER` 表用于维护用户信息,`ACT_ID_GROUP` 表用于维护组数据,`ACT_ID_MEMBERSHIP` 表用于维护用户与组之间的关系。引擎默认只支持用户或者组,但是通常情况下需要用到用户(人)、角色、部门等,换言之必须使 Activiti 支持多维度的配置,并非单维度的,这时就需要考虑如何扩展 Activiti 的功能。谨记尽量使用扩展源码的方式而并非直接修改源码的方式。

16.7.1 接管 Activiti 映射文件

由于 Activiti 封装了 MyBatis 框架,因此开发人员最好接管这些映射文件,首先自定义流程引擎配置类如代码清单 16-35 所示。

代码清单 16-35 `ShareniuProcessEngineConfiguration.java`

```
1 public class ShareniuProcessEngineConfiguration extends StandaloneProcessEngineConfiguration {
2     protected InputStream getMyBatisXmlConfigurationStream() {
3         return getResourceAsStream("com/shareniu/chapter16/identity/mappings.xml");
4     }
5 }
```

在上述代码中,第 3 行定义并返回了 MyBatis 映射文件的位置。直接将 Activiti 中需要使用到的 `mappings.xml`、`HistoricIdentityLink.xml`、`IdentityLink.xml` 复制一份放到

identity 目录下,如图 16-26 所示。

identity

impl

ShareniuProcessEngineConfiguration.java

activiti.cfg.xml

HistoricIdentityLink.xml

IdentityLink.xml

mappings.xml

图 16-26 配置文件目录

上述工作完毕需要在 mappings.xml 文件中引入 HistoricIdentityLink.xml、IdentityLink.xml 映射文件,如代码清单 16-36 所示。

代码清单 16-36 mappings.xml

```
1 <mapper resource="com/shareniu/chapter16/identity/IdentityLink.xml" />
2 <mapper resource="com/shareniu/chapter16/identity/HistoricIdentityLink.xml" />
3 <mapper resource="org/activiti/db/mapping/entity/HistoricIdentityLink.xml" />
4 <mapper resource="org/activiti/db/mapping/entity/IdentityLink.xml" />
5 ...//省略其他的映射文件
```

第 1~2 行对应上图中的映射文件,第 3~4 行为默认映射文件需要将其删除,如果不将上述两个默认配置文件删除,则引擎启动时报错。该操作完毕 Activiti 所有的映射文件已经被开发人员接管了。在实际项目开发中,如果觉得映射文件中的 SQL 不够灵活可直接在映射文件中进行修改。

16.7.2 禁用用户表和组表

如果 ACT_ID_USER、ACT_ID_GROUP、ACT_ID_MEMBERSHIP 表不能满足开发需求,则可以将这三张表禁用,具体实现如代码清单 16-37 所示。

代码清单 16-37 activiti.cfg.xml

```
1 <bean id="processEngineConfiguration"
2 class="com.shareniu.chapter16.identity.ShareniuProcessEngineConfiguration">
3 <property name="dbIdentityUsed" value="false" />
4 </bean>
```

通过设置 dbIdentityUsed 开关属性值为 false 即可,这样引擎启动时不会创建上述三张表。

16.7.3 自定义用户角色和部门表

本案例中需要使用到的表有用户表(SHARENIU_ID_USER)、角色表(SHARENIU_ID_ROLE)、部门表(SHARENIU_ID_DEP)、用户角色表(SHARENIU_ROLE_MEMBERSHIP)、用户部门表(SHARENIU_DEP_MEMBERSHIP),在使用之前需要创建这些表以及初始化表中数据如代码清单 16-38 所示。

代码清单 16-38 init.sql

```

1 CREATE TABLE SHARENIU_ID_USER (/* 用户 */
2   ID_ VARCHAR(64), /* 用户唯一标示 */
3   REV_ INTEGER, /* 版本号 */
4   NAME_ VARCHAR(255), /* 用户名 */
5   PWD_ VARCHAR(255), /* 密码 */
6   PRIMARY KEY (ID_)
7 );
8 CREATE TABLE SHARENIU_ID_DEP (/* 部门 */
9   ID_ VARCHAR(64), /* 部门唯一标示 */
10  NAME_ VARCHAR(255), /* 部门名称 */
11  PRIMARY KEY (ID_)
12 );
13 CREATE TABLE SHARENIU_DEP_MEMBERSHIP (/* 用户部门中间表 */
14   DEP_ID_ VARCHAR(64), /* 部门 id */
15   USER_ID_ VARCHAR(255), /* 用户 id */
16   PRIMARY KEY (DEP_ID_, USER_ID_)
17 );
18 CREATE TABLE SHARENIU_ID_ROLE (/* 角色 */
19   ID_ VARCHAR(64), /* 角色 id */
20   NAME_ VARCHAR(255), /* 角色名称 */
21   PRIMARY KEY (ID_)
22 );
23 CREATE TABLE SHARENIU_ROLE_MEMBERSHIP (/* 用户角色中间表 */
24   ROLE_ID_ VARCHAR(64), /* 角色 id */
25   USER_ID_ VARCHAR(255), /* 用户 */
26   PRIMARY KEY (ROLE_ID_, USER_ID_)
27 );
28 INSERT INTO SHARENIU_ID_USER (ID_) VALUES('shareniu1_1');
29 INSERT INTO SHARENIU_ID_USER (ID_) VALUES('shareniu1_2');
30 INSERT INTO SHARENIU_ID_USER (ID_) VALUES('shareniu1_3');
31 INSERT INTO SHARENIU_ID_USER (ID_) VALUES('shareniu1_4');
32 INSERT INTO SHARENIU_ID_USER (ID_) VALUES('shareniu1_5');
33 INSERT INTO SHARENIU_ID_ROLE (ID_, NAME_) VALUES('role1', 'role1');
34 INSERT INTO SHARENIU_ID_ROLE (ID_, NAME_) VALUES('role2', 'role2');
35 INSERT INTO SHARENIU_ROLE_MEMBERSHIP (ROLE_ID_, USER_ID_) VALUES('role1', 'shareniu1_1');
36 INSERT INTO SHARENIU_ROLE_MEMBERSHIP (ROLE_ID_, USER_ID_) VALUES('role2', 'shareniu1_2');
37 INSERT INTO SHARENIU_ID_DEP (ID_, NAME_) VALUES('shareniu1', 'shareniu1_1');
38 INSERT INTO SHARENIU_ID_DEP (ID_, NAME_) VALUES('shareniu2', 'shareniu1_2');
39 INSERT INTO SHARENIU_DEP_MEMBERSHIP (DEP_ID_, USER_ID_) VALUES('shareniu1', 'shareniu1_3');
40 INSERT INTO SHARENIU_DEP_MEMBERSHIP (DEP_ID_, USER_ID_) VALUES('shareniu2', 'shareniu1_4');
41 INSERT INTO SHARENIU_DEP_MEMBERSHIP (DEP_ID_, USER_ID_) VALUES('shareniu2', 'shareniu1_5');

```

16.7.4 扩展任务节点参与者表

ACT_RU_IDENTITYLINK 表可以存储任务节点的候选人或者候选组,在使用时可以将该表中的 GROUP_ID_ 字段存储角色 id,这样人员与角色的问题就解决了,那么怎么存储

部门 id 呢? 很显然需要为该表增加一个字段以存储部门 id 值, 执行 SQL 如代码清单 16-39 所示。

代码清单 16-39 init.sql

```
1 ALTER TABLE ACT_RU_IDENTITYLINK ADD COLUMN DEP_ID VARCHAR(255);
2 ALTER TABLE ACT_HI_IDENTITYLINK ADD COLUMN DEP_ID VARCHAR(255);
```

16.7.5 自定义任务节点参与者命令类

IdentityLinkEntity 类为 ACT_RU_IDENTITYLINK 表对应的映射实体类, 该类中并没有定义部门 id 属性, 所以需要考虑如何扩展 IdentityLinkEntity 类, 首先定义一个类并继承该类, 如代码清单 16-40 所示。

代码清单 16-40 SharenuiIdentityLinkEntity.java

```
1 public class SharenuiIdentityLinkEntity extends IdentityLinkEntity {
2     protected String depId; //部门 id
3     public Object getPersistentState() {
4         Map<String, Object> persistentState = (Map<String, Object>) super.getPersistentState();
5         if (this.depId != null) {
6             persistentState.put("depId", this.depId);
7         }
8         return persistentState;
9     }
10 }
```

第 3~9 行重写了父类中的 getPersistentState 方法, 并添加了 depId 属性值。第 15.6 节讲解了 Activiti 生成 SQL 执行 id 值的规则, 如果要想操作 SharenuiIdentityLinkEntity 类, 那么对于该类的 SQL 执行 id 值(插入操作)应该是 insertSharenuiIdentityLink, IdentityLink.xml 文件中新增 SQL, 如代码清单 16-41 所示。

代码清单 16-41 IdentityLink.xml

```
1 <insert id="insertSharenuiIdentityLink">
2     parameterType="com.sharenui.chapter16.identity.SharenuiIdentityLinkEntity">
3         insert into ${prefix}ACT_RU_IDENTITYLINK (ID_, REV_, TYPE_, USER_ID_, GROUP_ID_, TASK_ID_,
4         PROC_INST_ID_, PROC_DEF_ID_, DEP_ID_) values (# {id, jdbcType = VARCHAR},
5             1,
6             # {type, jdbcType = VARCHAR},
7             # {userId, jdbcType = VARCHAR},
8             # {groupId, jdbcType = VARCHAR},
9             # {taskId, jdbcType = VARCHAR},
10            # {processInstanceId, jdbcType = VARCHAR},
11            # {processDefId, jdbcType = VARCHAR},
12            # {depId, jdbcType = VARCHAR})
13 </insert>
```

为了保持原有功能可以正常使用,不要把 SQL 执行 id 值为"insertIdentityLink"的标签删除。下面定义一个命令类执行上述 SQL,如代码清单 16-42 所示。

代码清单 16-42 AddIdentitylinkCmd.java

```
1 public class AddIdentitylinkCmd implements Command<Void> {
2     private String depId; //部门 id 省略构造方法
3     private String taskId; //任务 id
4     public Void execute(CommandContext commandContext) {
5         ShareniuIdentityLinkEntity identityLinkEntity = new ShareniuIdentityLinkEntity();
6         identityLinkEntity.setType(IdentityLinkType.CANDIDATE);
7         identityLinkEntity.setDepId(depId);
8         identityLinkEntity.setTaskId(taskId);
9         identityLinkEntity.insert();
10        return null;
11    }
12 }
```

定义一个测试类如代码清单 16-43 所示。

代码清单 16-43 App.java

```
1 public void executeCommand() {
2     processEngine.getManagementService().executeCommand(new AddIdentitylinkCmd("sharenui1",
3     null));
4 }
```

执行上述代码,ACT_RU_IDENTITYLINK 表新增的数据如图 16-27 所示。

ID	REV	GROUP_ID	TYPE	DEP_ID
25001	1	(Null)	candidate	sharenui1

图 16-27 ACT_RU_IDENTITYLINK 表新增的数据

扩展

可结合该案例将数据同步到 ACT_HI_IDENTITYLINK 表中。

16.7.6 流程文档支持设置部门属性

定义一个流程文档如代码清单 16-44 所示。

代码清单 16-44 usertaskactivitybehavior.bpmn20.xml

```
1 <process id = "dep" name = "dep" isExecutable = "true">
2     <startEvent id = "start" name = "Start"></startEvent>
3     <userTask id = "usertask1" name = "usertask1">
4         activiti:candidateGroups = "role1" sharenui:depId = "sharenui1,sharenui2"></userTask>
5     <sequenceFlow id = "flow1" sourceRef = "start" targetRef = "usertask1"></sequenceFlow>
6     <userTask id = "usertask2" name = "usertask2"></userTask>
```

```

7 <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="usertask2"></sequenceFlow>
8 <endEvent id="endevent1" name="End"></endEvent>
9 <sequenceFlow id="flow3" sourceRef="usertask2" targetRef="endevent1"></sequenceFlow>
10 </process>

```

第4行 shareniu:depId 属性定义了部门信息,关于任务节点属性的解析可以参考第14.4节的讲解。接下来需要分三步走,首先需要解析该属性值,其次当流程实例运转到 usertask1 任务节点时,需要获取该属性解析后的结果并进行相应操作,最后将获取到的数据添加到 ACT_HI_IDENTITYLINK 表中。

16.7.7 解析及运用流程文档部门属性

自定义 ShareniuUserTaskActivityBehavior 类并继承 UserTaskActivityBehavior 类,该类的核心定义如代码清单 16-45 所示。

代码清单 16-45 ShareniuUserTaskActivityBehavior.java

```

1 public class ShareniuUserTaskActivityBehavior extends UserTaskActivityBehavior {
2     ...//省略构造方法
3     protected void handleAssignments(Expression assigneeExpression,
4         Expression ownerExpression,
5         Set<Expression> candidateUserExpressions,
6         Set<Expression> candidateGroupExpressions, TaskEntity task,
7         ActivityExecution execution) {
8         super.handleAssignments(assigneeExpression, ownerExpression,
9             candidateUserExpressions, candidateGroupExpressions, task, execution);
10        PvmActivity activity = execution.getActivity();
11        ManagementService managementService = execution.getEngineServices().getManagementService();
12        Object property = activity.getProperty("shareniuExt"); // 获取 shareniuExt 属性值
13        Map<String, List<ExtensionAttribute>> extensionAttribute = null;
14        if (property != null) {
15            extensionAttribute = (Map<String, List<ExtensionAttribute>>) property;
16        }
17        List<ExtensionAttribute> list1 = extensionAttribute.get("depId");
18        for (ExtensionAttribute extensionAttribute2 : list1) {
19            String value = extensionAttribute2.getValue();
20            String[] split = value.split(",");
21            for (String depId : split) {
22                managementService.executeCommand(new AddIdentitylinkCmd(depId, task.getId()));
23            }
24        }
25    }
26 }

```

第8行直接调用父类中的 handleAssignments 进行处理,第12行获取 shareniuExt 属性值,关于该属性的获取可以参考第14.4节,第17行获取 depId,第19~24行循环遍历配置的部门信息,并将其添加到 ACT_RU_IDENTITYLINK 表中。

接下来查看 activiti.cfg.xml 文件内容,如代码清单 16-46 所示。

代码清单 16-46 activiti.cfg.xml

```

1 <bean id="processEngineConfiguration"
2   class="com.shareniu.chapter16.identity.ShareniuProcessEngineConfiguration">
3   <property name="dbIdentityUsed" value="false"/>
4   <property name="customDefaultBpmnParseHandlers">
5     <list>
6       <bean class="com.shareniu.chapter14.ShareniuUserTaskParseHandler"></bean>
7     </list>
8   </property>
9   <property name="activityBehaviorFactory" ref="shareniuActivity" />
10 </bean>
11 <bean id="shareniuActivity"
12   class="com.shareniu.chapter16.identity.ShareniuActivityBehaviorFactory" />

```

上述代码中,第 6 行和第 11~12 行定义类可以参考第 14.4 节。

部署代码清单 16-44 定义的流程文档并启动流程实例,ACT_RU_IDENTITYLINK 表新增的数据如图 16-28 所示。

ID_	REV_	GROUP_ID_	TYPE_	DEP_ID_	TASK_ID_
67505	1	role1	candidate	(Null)	67504
67506	1	(Null)	candidate	shareniu1	67504
67507	1	(Null)	candidate	shareniu2	67504

图 16-28 ACT_RU_IDENTITYLINK 表新增的数据

16.7.8 自定义代办 SQL

ACT_RU_IDENTITYLINK 表中已经成功添加了 GROUP_ID_ 为 role1(对应的人员 id 有 shareniu1_1),DEP_ID_ 为 shareniu1(对应的人员有 shareniu1_3)和 shareniu2(对应的人员有 shareniu1_4)三条数据,这时如果使用 Activiti 本身的用户代办任务查询功能,很显然是查询不到的,因此接下来需要书写符合自身业务需求的查询代办 SQL 语句,如代码清单 16-47 所示。

代码清单 16-47 init.sql

```

1 SELECT t.ID_,t.NAME_
2 FROM ACT_RU_TASK AS t
3 RIGHT JOIN
4   (SELECT DISTINCT *
5     FROM
6       (SELECT r.TASK_ID_,
7         sir.USER_ID_
8       FROM
9         (SELECT GROUP_ID_,
10          TASK_ID_
11        FROM ACT_RU_IDENTITYLINK I,

```

```

12      ACT_RU_TASK T
13      WHERE I.TASK_ID IS NOT NULL
14      AND I.GROUP_ID IS NOT NULL
15      AND I.TASK_ID = T.ID_
16      AND T.ASSIGNEE IS NULL
17      AND TYPE_ = 'candidate') r
18      INNER JOIN SHARENIU_ROLE_MEMBERSHIP sir ON r.GROUP_ID = sir.ROLE_ID_
19      UNION SELECT r.TASK_ID_,
20      sdm.USER_ID_
21      FROM
22      (SELECT DEP_ID_,
23      TASK_ID_
24      FROM ACT_RU_IDENTITYLINK I,
25      ACT_RU_TASK T
26      WHERE I.TASK_ID IS NOT NULL
27      AND I.DEP_ID IS NOT NULL
28      AND I.TASK_ID = T.ID_
29      AND T.ASSIGNEE IS NULL
30      AND TYPE_ = 'candidate') r
31      INNER JOIN SHARENIU_DEP_MEMBERSHIP sdm ON r.DEP_ID = sdm.DEP_ID_
32      UNION SELECT TASK_ID_,
33      USER_ID_
34      FROM ACT_RU_IDENTITYLINK I,
35      ACT_RU_TASK T
36      WHERE TASK_ID IS NOT NULL
37      AND USER_ID IS NOT NULL
38      AND I.TASK_ID = T.ID_
39      AND T.ASSIGNEE IS NULL
40      AND TYPE_ = 'candidate') t
41      WHERE t.USER_ID_ = 'sharenui_4') AS r ON t.ID_ = r.TASK_ID_ HI

```

其中,第41行的 USER_ID_ 值为用户 id,运行上述 SQL 语句,查询的结果集集如图 16-29 所示。

ID_	NAME_
67504	usertask1

图 16-29 ACT_RU_IDENTITYLINK 表新增的数据

16.8 接管 Activiti 实体管理类

以修复 Activiti 中 ACT_HI_PROCINST 不支持 Juel 表达式为例,详细讲解如果接管 Activiti 中的实体管理类,如果开发人员不打算使用 Activiti 的用户表、组表或者用户组表,可结合该案例自行实现。

首先定义一个流程文档如代码清单 16-48 所示。

代码清单 16-48 taskJuel.bpmn

```

1 <process id="juel" name="juel" isExecutable="true">
2   <startEvent id="startevent1" name="Start"></startEvent>
3   <userTask id="usertask1" name="$ {a}"></userTask>
4   <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"></sequenceFlow>
5   <endEvent id="endevent1" name="End"></endEvent>
6   <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="endevent1"></sequenceFlow>
7 </process>

```

第 3 行使用了 `${a}` 表达式作为 id 为 `usertask1` 任务节点的名称。部署上述流程文档并在启动流程实例时设置变量 `a` 的值为 `"shareniu"`，如代码清单 16-49 所示。

代码清单 16-49 taskJuel.bpmn

```

1 public void startProcessInstanceId() throws IOException {
2   Map<String, Object> var = new HashMap<String, Object>();
3   var.put("a", "shareniu");
4   runtimeService.startProcessInstanceId("juel:1:4", var);
5 }

```

ACT_HI_PROCINST 表中新增数据如图 16-30 所示。

ID	PROC_DEF_ID	ACT_NAME	ACT_ID	ACT_TYPE
5002	juel:1:4	Start	startevent1	startEvent
5004	juel:1:4	<code>\${a}</code>	usertask1	userTask

图 16-30 ACT_HI_PROCINST 表新增的数据

图 16-30 ACT_HI_PROCINST 表中 ID 值为 5004 的 ACT_NAME 字段值依然为 `${a}`，这很显然是 Activiti 中的一个 Bug 导致的。

自定义一个实体管理类并继承 `DefaultHistoryManager` 如代码清单 16-50 所示。

代码清单 16-50 ShareniuHistoryManager.java

```

1 public class ShareniuHistoryManager extends DefaultHistoryManager {
2   public void recordActivityStart(ExecutionEntity executionEntity) {
3     if (isHistoryLevelAtLeast(HistoryLevel.ACTIVITY)) {
4       if (executionEntity.getActivity() != null) {
5         ExpressionManager expressionManager = Context
6           .getProcessEngineConfiguration().getExpressionManager();
7         Expression exp = expressionManager.createExpression((String) executionEntity
8           .getActivity().getProperty("name"));
9         IdGenerator idGenerator = Context.getProcessEngineConfiguration().getIdGenerator();
10        String processDefinitionId = executionEntity.getProcessDefinitionId();
11        String processInstanceId = executionEntity.getProcessInstanceId();
12        String executionId = executionEntity.getId();
13        HistoricActivityInstanceEntity historicActivityInstance = new
14          HistoricActivityInstanceEntity();
15        historicActivityInstance.setId(idGenerator.getNextId());

```

```

16 historicActivityInstance.setProcessDefinitionId(processDefinitionId);
17 historicActivityInstance.setProcessInstanceId(processInstanceId);
18 historicActivityInstance.setExecutionId(executionId);
19 historicActivityInstance.setActivityId(executionEntity.getActivityId());
20 historicActivityInstance.setActivityName((String)exp.getValue(executionEntity));
21 historicActivityInstance.setActivityType((String)executionEntity.getActivity()
22     .getProperty("type"));
23 historicActivityInstance.setStartTime(Context.getProcessEngineConfiguration().getClock()
24     .getCurrentTime());
25 if (executionEntity.getTenantId() != null) {
26     historicActivityInstance.setTenantId(executionEntity
27         .getTenantId());
28 }
29 getDbSqlSession().insert(historicActivityInstance);
30 ...//转发 HISTORIC_ACTIVITY_INSTANCE_CREATED 事件
31 }
32 }
33 }
34 }

```

在上述代码中,第5~6行获取表达式管理器,第7~8行根据 name 创建 Expression 实例对象,第20行计算表达式的值。

自定义实体管理工厂类并继承 DefaultHistoryManagerSessionFactory,如代码清单 16-51 所示。

代码清单 16-51 ShareniuProcessHistoryManagerSessionFactory.java

```

1 public class ShareniuProcessHistoryManagerSessionFactory extends、
2     DefaultHistoryManagerSessionFactory {
3     public Session openSession() {
4         return new ShareniuHistoryManager();
5     }
6 }

```

第4行直接实例化 ShareniuHistoryManager 类。

将自定义实体管理工厂类注入流程引擎配置类如代码清单 16-52 所示。

代码清单 16-52 activiti.cfg.xml

```

1 <bean id="processEngineConfiguration"
2     class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
3     <property name="customSessionFactories">
4         <list>
5             <bean class="com.shareniu.chapter16.ShareniuProcessHistoryManagerSessionFactory">
6                 </list>
7             </property>
8         </bean>

```


通过 customSessionFactory 开关属性将 ShareniuProcessHistoryManagerSessionFactory 类注入流程引擎配置类。再次启动流程实例,ACT_HI_PROCINST 表中新增的数据如图 16-31 所示。

ID	PROC DEF ID	ACT NAME	ACT ID	ACT TYPE
5002	juel:1:4	Start	startevent1	startEvent
5004	juel:1:4	\$(a)	usertask1	userTask
7502	juel:1:4	Start	startevent1	startEvent
7504	juel:1:4	shareniu	usertask1	userTask

图 16-31 ACT_HI_PROCINST 表新增的数据

新增的数据表达式已经被成功解析了。

在以上代码中,第 5~6 行获取表达式管理器,第 7~8 行根据 name 创建 Expression 类实例,第 9 行计算表达式值。图 16-31 中新增数据中 ACT_HI_PROCINST 表中新增数据 16-31 单看代码,代码清单 16-31

所示。

代码清单 16-31 ShareniuProcessHistoryManagerSessionFactory.java

1 public class ShareniuProcessHistoryManagerSessionFactory extends
2 DefaultHistoryManagerSessionFactory {

3 public Session openSession() {

4 return new ShareniuHistoryManager();

5 }

6 }

7 }

8 }

9 }

10 }

11 }

12 }

13 }

14 }

15 }

16 }

17 }

18 }

19 }

20 }

21 }

22 }

本书特色

结构清晰。本书由浅入深、以点带面，采用从整体到局部，再从局部放眼全局的视角，力求展示Activiti框架的全貌。

原理结合实战。本书不甘心仅仅追求原理的讲解，更希望通过原理延伸到实际项目开发中具体问题的解决，从而达到学以致用目的。

通俗易懂。本书依然服务于广大的Activiti初学者以及爱好者，尽量避免过于理论的描述方式，尽量做到少盲点、无盲点，从而更加“享受式”的学习，语言浅显易懂而不失专业。

技术全面。本书分析讲解了80%以上的源码，并对框架级别的Bug和缺陷大胆假设，小心取证，使读者对Activiti框架有一个全面、全新的认识。

代码移植性高。本书的案例来自真实的企业级应用，代码经过工业环境的验证和实践、可移植性强、可塑性高、力争追求“小”而精。

扫一扫



课件下载、样书申请
教材推荐、技术交流

上架建议：人工智能/自运化/概率

ISBN 978-7-302-47498-2



9 787302 474982 >

定价：79.00元